Humboldt-Universität zu Berlin

Institut für Informatik

Lehrstuhl für Rechnerorganisation und Kommunikation

Diplomarbeit

CollabKit – A Multi-User Multicast Collaboration System based on VNC

Christian Beier

19. April 2011

Gutachter

Prof. Dr. Miroslaw Malek

Prof. Dr. Jens-Peter Redlich

Betreuer

Peter Ibach <ibach@informatik.hu-berlin.de>

Abstract

Computer-supported real-time collaboration systems offer functionality to let two or more users work together at the same time, allowing them to jointly create, modify and exchange electronic documents, use applications, and share information location-independently and in real-time.

For these reasons, such collaboration systems are often used in professional and academic contexts by teams of knowledge workers located in different places. But also when used as computer-supported learning environments – electronic classrooms – these systems prove useful by offering interactive multi-media teaching possibilities and allowing for location-independent collaborative learning.

Commonly, computer-supported real-time collaboration systems are realised using remote desktop technology or are implemented as web applications. However, none of the examined existing commercial and academic solutions were found to support concurrent multi-user interaction in an application-independent manner. When used in low-throughput shared-medium computer networks such as WLANs or cellular networks, most of the investigated systems furthermore do not scale well with an increasing number of users, making them unsuitable for multi-user collaboration of a high number of participants in such environments.

For these reasons this work focuses on the design of a collaboration system that supports *concurrent multi-user interaction* with standard desktop applications and is able to serve a high number of users on low-throughput shared-medium computer networks by making use of *multicast data transmission*.

The developed multi-user multicast collaboration system named CollabKit, realised by integrating and extending existing technologies, was compared against a conventional unicast remote desktop system and found to significantly outperform it when several clients needed to be served. Regarding the functionality requirements and performance metrics defined in this work, CollabKit could achieve the expected results.

This work shows that it is possible to create a computer-supported real-time collaboration system with multi-user and multicast support by integrating existing technologies and extending them with custom implementations where needed: The developed system supports application-independent concurrent operation by multiple users, per-user graphical annotations and window sharing and scales well with an increasing number of users.

Contents

1 Introduction

Collaboration means working together. Computer-supported real-time collaboration systems allow two or more participants to work together simultaneously. Being *computer-supported*, they provide the advantage of being able to make electronic documents, multimedia content or interactive applications available to participants, independent of their location. The *real-time* properties of such systems enable users to concurrently ask and answer questions, brainstorm, and thus to rapidly draw, refuse, or accept conclusions. Done asynchronously, such activities would take a much longer time.

These characteristics make computer-supported real-time collaboration systems useful in both professional as well as educational contexts. On the one hand, they enable knowledge workers and scientists to exchange information and to jointly create, share and modify electronic artefacts. On the other hand, computer-supported real-time collaboration systems can be used for electronic teaching and learning: by employing such *electronic classrooms* for education, traditional teaching material can be made more interactive, abstract concepts can be visualised using simulations and multimedia content can be incorporated into the teaching process, allowing participants to interact with each other and the provided material.

1.1 Problem Statement

Lack of Fully Concurrent Multi-User Operation

The first area in which common computer-supported real-time collaboration systems are limited though is support of fully concurrent *multi-user* interaction:

On the one hand, there is one class of collaboration systems that does support fully concurrent user interaction, but such systems are confined to one or a few built-in applications specifically designed for that system with multi-user support in mind. They do not allow users to interact with unmodified standard desktop applications.

On the other hand, the second class of computer-supported real-time collaboration systems does allow participants to use any kind of desktop application, but they only support user interaction in a turn-taking mode. Only one user at a time can be in control of the shared desktop, there is only sequential but no concurrent interaction.

Bad Scalability in Low-Throughput Shared-Medium Networks

The aspect of multi-user support inevitably leads to the second area in which existing systems have shortcomings: when sharing applications or whole desktops – especially on low-throughput computer networks characterised by shared medium access such as wireless local area networks – the system's user-perceived performance degrades with an increasing number of connected users. This is because the same data is sent to each and

every user individually: the more users are connected, the less throughput capacity is available to each one.

1.2 Approach

In order to address the first problem – lack of fully concurrent multi-user operation in existing systems – the first focal point of this work was to **create a computer-supported real-time collaboration system with support for fully concurrent multi-user operation**. The eventual goal was to develop, implement, and test an easy-to-use collaboration system that allows its users to *simultaneously* interact with *any kind* of application on a shared desktop. To achieve this, existing technologies were examined and suitable ones integrated to form a collaboration system with the desired features.

The second problem – bad scalability in low-throughput networks – could not be solved by simply integrating existing technologies though. Instead, this required enhancing the way data representing shared applications is delivered to the system's users. This meant designing and implementing an extension of an existing remote desktop technology that would make data transmissions use the shared medium more efficiently. The chosen approach to accomplish this was to **fit the created system with support for multicast data transmission**. This allows a high number of users to efficiently use the created collaboration system on a low-throughput shared-medium network.

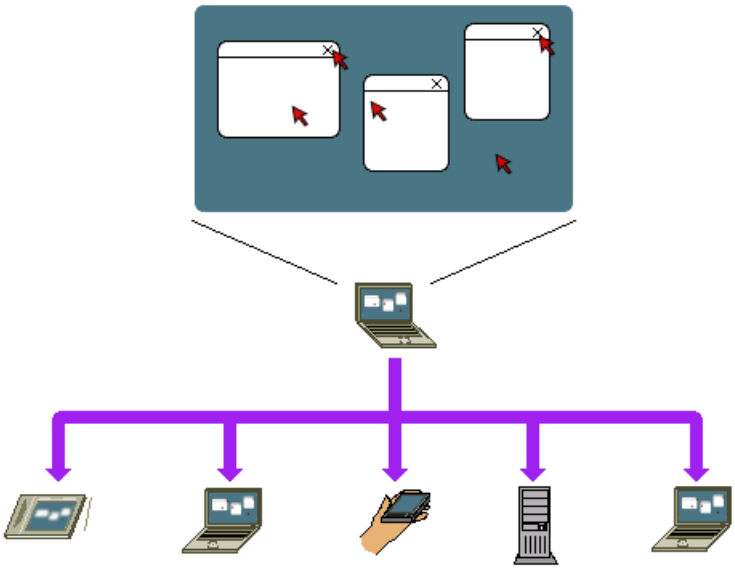


Figure 1: A computer-supported real-time collaboration system that supports *concurrent* multi-user interaction and transmits the shared desktop *once* to *all* clients using multicast.

1.3 Structure of this Work

First specific use case scenarios of computer-supported real-time collaboration systems are analysed with regard to the functional and non-functional requirements they pose to the system in use. Then a survey of existing systems and basic technology is done, analysing conformance to the identified requirements. This includes common remote desktop technologies like the X Window System, Virtual Network Computing and the Remote Desktop Protocol as well as other commercial and academic solutions.

Using the findings of these investigations, the design of a computer-supported real-time collaboration system that supports fully concurrent multi-user interaction and multicast data transmission is presented.

Then, the implementation process of the devised system is documented and the system itself is evaluated with regard to its conformance to the initial requirements. A summary of findings regarding the work done and a look-out to possible future work that can build upon the developed system mark the conclusion of this work.

2 Real-Time Collaboration Use Cases and Requirements

There are a multitude of application possibilities for real-time collaboration systems. The benefits they provide can come in handy not only when used as *electronic classrooms* in teaching environments, but such systems can also be valuable in professional contexts in which a team of knowledge workers uses and edits electronic documents *together*. This section presents the most fundamental use cases ranging from simple single-user presentation scenarios to collaborative work settings with several participants.

2.1 Use Cases

2.1.1 Presentations

Presentations using a portable computer and a projector nowadays are quite common in areas like school, university or industry. The setting is always quite similar: The presenter stands in front of the audience and shows something to them on the projector's screen, putting across some idea using simple slides or demonstrating a running application. In this scenario, the audience remains more or less *passive*.

A common problem for the presenter is how to flip back and forth through the slides quickly. When no remote control is available, presenters always have to get back to their computer in order to turn pages. A similar problem occurs when an application is to be demonstrated: presenters have to stop explaining something in front of the audience and have to return to their computer to operate the program. Doing so, interaction with the audience, like responding to remarks or hints, is hampered. Figure 2 illustrates this.

Figure 2: Typical problem while presenting: explaining something to the audience and operating the presentation equipment have to happen concurrently [100].

There are two solutions to these problems: first, presenters could use a small and handy device like a sub-notebook or a smartphone. This would allow them to control the presentation and still be in close contact with the audience. However, operating a complex application with such a device can be cumbersome. The other solution is to leave presenters seated behind their large, easy-to-use notebook and to provide them with extended means of presentation that go beyond the usual mouse cursor, like the possibility of making graphical annotations on-screen in order to highlight some region of interest. This probably is the better approach for a task like application demonstration.

Finally, it is very beneficial for the audience if they can somehow interact with what is shown on the projector's screen. Within a traditional presentation setup, only verbal interaction is easy. Drawing attention to some special region on the screen just by verbal means can be difficult, though. One the other hand, getting up and having to walk to the front to highlight something on the screen is as equally cumbersome. It is a big advantage for the audience to be able to *directly* draw graphical annotations onto the projector's display from where they are or to show applications running on *their* computers on the projector's screen.

2.1.2 Electronic Teaching

The electronic teaching scenario, as seen here, is a situation in which a relatively small group of instructors and learners is intensively engaged in creating comprehension of some matter. Teaching materials can be simple text documents, websites or multimedia contents like audio and video files. In fact, the essential benefit of real-time collaboration systems used as electronic classrooms over more traditional ways of teaching is that animated electronic content can be shown and directly interacted with. Training the use of some complex program is a good example: instructor and learners can explore the usage of the application *together*.

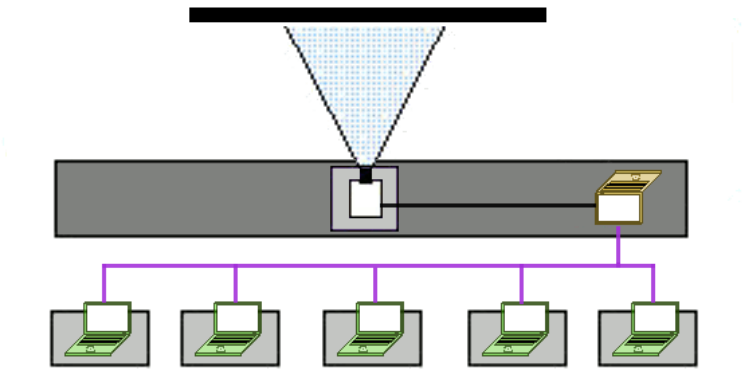


Figure 3: An electronic classroom scenario: students (green) can work together on the yellow computer that is optionally connected to a projector.

When students have (limited) access to the instructor's desktop, they can ask questions regarding some specific parts of the user interface a lot more easily and exactly. On the other hand, if problems arise when the assignment was to carry out a specific task with the program by oneself, it can be very helpful to show one's own desktop to the others.

In another slightly different scenario, illustrated by figure 3, students can work together on a single desktop to collaboratively solve assigned tasks or to learn how to use some program *together*, assisting and guiding each other. Such a multi-user approach to electronic classrooms employs the benefits of collaborative learning [88]: By allowing students to *jointly* interact with complex objects on the screen, new ways of learning and understanding [104, p. 132][118, p. 7] can be developed.

A third, to some extent educational scenario is that of an assistance system. In this use case, a computer user interface is normally operated by one user alone. Only if the user is in need of help, an assistant can join in so that the two operate the user interface together. This provides the assistant with a far more effective way of guiding the user than, for instance, support by phone can offer.

2.1.3 Professional Collaboration

Professional collaboration scenarios are situations in which a group of people work together trying to solve a problem. This can include joint brainstorming, collecting relevant bits of information, compiling these bits into documents and creating a solution using these documents – all done in teamwork, collaboratively.

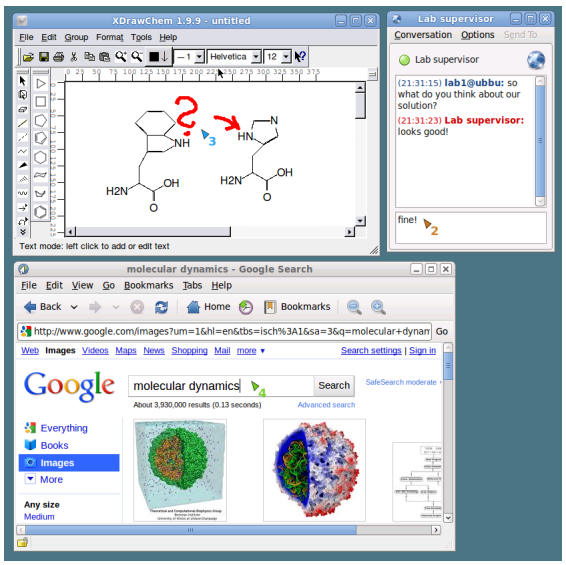


Figure 4: A scientific professional collaboration scenario with three participants, each operating a different application.

It can be that all participants are in the same room, on the other hand it is possible that they communicate from different parts of the world. The professional collaboration use case also differs from the electronic classroom use case in that there is no instructor-student hierarchy: in a professional collaboration scenario, participants are not being taught but are equal, working together. There is much more emphasis on collaborative work.

There are lots of possible fields of application for real-time collaboration systems. For instance, scientists can use it for collaborative brainstorming or developing of ideas, very much like a shared whiteboard, but rather an electronic one with multimedia objects on it. Such a system can also be of use for the industry: in business contexts, it is quite common that considerably large documents have to be created, often containing multimedia content and requiring teamwork to be compiled. Especially for geographically distributed teams, a real-time collaboration system used for *computer supported collaborative work* can be beneficial [105], even more if it allows fully concurrent user interaction [113].

2.2 Requirements Analysis

Taking the aforementioned use case scenarios as a basis, it is possible to deduce requirements the used software has to meet. First, high-level *non-functional* requirements specifying a system's general characteristics and overall qualities [102, p. 187] are dealt with. Then, lower-level *functional* requirements which define particular abilities and functions of a system [102, p. 188] are investigated.

Naturally, requirements regarding software *functionality* differ the most between the considered use cases, whereas *non-functional* requirements specifying overall system qualities were found to be more uniform.

2.2.1 Non-Functional Requirements

Although the considered use cases differ in some particular aspects mostly regarding specific functionality, they all pose similar requirements when it comes to overall characteristics of the software used to operate a real-time collaboration system. Along with particular functionality offered, it is the fulfilment of these *non-functional* requirements that accounts for a good user experience.

Performance

First of all, all three use cases pose certain requirements regarding performance: especially for real-time collaboration, a slow system is a system nobody will like to use. When working together in real-time, there should not be a too long time span between a user action and the response triggered by that action: too high delays increase task completion time and user error rate [90]. Thus, the system should try to keep *latency* low. Additionally, in all three uses cases rather bulky image data representing a whole

desktop or single windows is transferred between participants. For a good user experience, this should happen as fast as possible, essentially meaning that the system in use should achieve high effective *throughput*.

Scalability

Furthermore, in all of the considered use cases the number of participants is not predetermined: it can be that just two or well over twenty users take part. Ideally, the used system should deliver the same high level of performance with any number of connected clients. Put short, it should do well in terms of *scalability*.

Usability

Then, it can be expected for every use case that the level of technical computer knowledge among users differs. Besides, while experts would be able use a system that is overly complicated to operate, that use would still be unnecessarily time-consuming. Therefore, *ease of use* is another important non-functional requirement.

Portability

Additionally, users of such a collaboration system do not only have different levels of computer knowledge, it is also very likely that they use different operating systems, at least in the presentation and professional collaboration use cases. The used client application, if any, should therefore be *platform-independent* or at least portable.

Security

Finally, *security* of the employed system is of importance. On the one hand, this includes the basic question of who is allowed to use the system and who is not. On the other hand, security includes guaranteeing confidentiality, availability and integrity of the communication that is taking place.

2.2.2 Functional Requirements

The following subsection deals with the *functional* requirements the software used to facilitate a real-time collaboration system has to meet. First, basic functional requirements pertaining to specific features and actual functionality of the used software are identified per use case and summed up. Then, those functional requirements are deduced whose fulfilment helps to satisfy the high-level non-functional requirements identified in subsection 2.2.1.

Basic Functional Requirements

Regarding Presentations. Like mentioned before, it can be quite beneficial in presentation scenarios if the audience is able to interact somehow with what is shown on the projector's screen, improving interaction between presenter and viewers. In terms of

functionality this means the audience should be able to remote control the presenter's desktop, i. e. to do basic things like moving windows or flipping back and forth in slides. But other more advanced features are conceivable as well: the ability to draw graphical annotations onto the presenter's desktop would be very useful to highlight certain regions of the screen for the others. Also, functionality to show one's own window on the presenter's desktop can be a helpful feature for an attendee of the presentation, for example to show a certain document or application to other participants.

Thus, functionality requirements for the presentation use case were specified as:

- functionality to remote control the presenter's desktop
- an annotation mode
- functionality for presentation viewers to export their own windows to the presenter's desktop

Regarding Electronic Teaching. For the electronic classroom use case, functionality requirements were found to be similar, with the exception that there is a stronger emphasis on window or desktop sharing in *both* directions, from instructor to students and from students to instructor. On the one hand, it should be possible for students to share their desktops (or parts thereof) to the instructor or to others in order to get specific help. On the other hand, students should be able to view the instructor's desktop, especially when they are remotely located. In addition, they should be able to exercise limited control on the instructor's desktop, for example in order to ask about the usage of some control element or to demonstrate something to other students. For this purpose, a graphical annotation mode on the instructor's desktop would be useful in this scenario as well.

Therefore the requirements regarding functionality for an electronic classroom use case are:

- functionality to view and limitedly remote control the instructor's desktop
- an annotation mode on the instructor's desktop
- functionality for students to export their own windows to the instructor's desktop

Regarding Professional Collaboration. The professional collaboration use case has functionality requirements that are almost the same as those of the presentation and electronic teaching scenarios. The one additional requirement the professional collaboration use case has is that the system should support several participants concurrently working together at one desktop. Since the fundamental means of controlling a desktop are mouse pointer and keyboard *and* because fully concurrent collaboration provides advantages over turn-taking [113], the system thus ideally should provide every participant with their *own* mouse cursors and keyboard foci. Such a computer supported real-time collaboration system allows users to interact with objects on the desktop *jointly* and *simultaneously*.

Functionality requirements for professional collaboration use can thus be summed up as:

- functionality to view and remote control the shared desktop
- a graphical annotation mode on the shared desktop
- functionality for participants to export their own windows to the shared desktop
- support of multiple mouse cursors and keyboard foci on the shared desktop

Functional Requirements Related to Non-Functional Ones

Related to Performance and Scalability. Since one of the main uses of the system is to transmit rather bulky image data, the underlying network's maximum throughput is likely to pose a fundamental constraint. For some of the considered use cases, it is reasonable to expect wireless LANs with 54 MBit/s (802.11a/g) or just 11 MBit/s (802.11b) gross data rate. With this in mind, it is obvious that for instance delivering 25 fps of uncompressed RGB data to multiple participants will quickly exhaust the network's capacity.

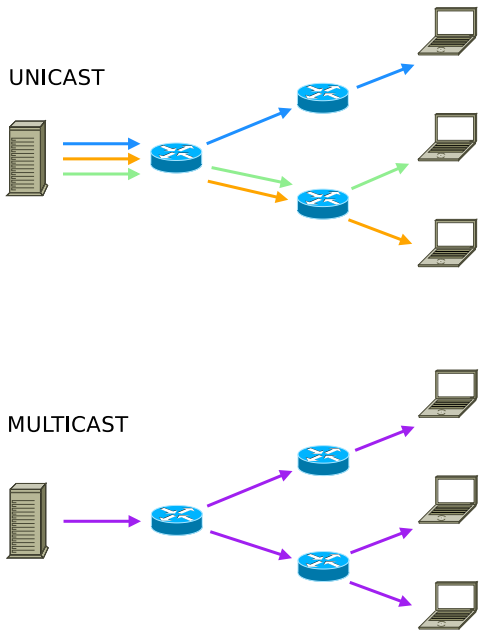


Figure 5: Multicast data transmission provides significant channel capacity savings compared to unicast.

Therefore, the maximum achievable data throughput of the underlying network was identified as the primary bottleneck regarding the system's scalability. Taking into account the considered use cases, it is reasonable to maintain that in most cases *multiple* users will be connected to the desktop they are jointly working on. Since the image data representing this remote desktop is the same for all connected participants, an

obvious approach to alleviate the constraints posed by limited network capacity is to use *multicast* data transmission instead of unicast. This way data just gets sent *once* to all connected users instead of being delivered to each and every one individually. It was concluded that by using multicast data transmission instead of unicast, the system's performance would not be worsened by an increasing number of participants anymore, as illustrated by figure 5. Thus, the use of **multicast transmission of image data** was made a fundamental requirement regarding the system's performance and scalability.

Regarding the maximum achievable **throughput** that can be measured at the client side, the following metrics show the advantage of multicast over unicast: For the *unicast* case, the maximum throughput observable by a client *cl* can be defined as

$$T_{cl} = \min\left(T_p, \frac{T_{sp}}{N_{sp}}\right) \tag{1}$$

In this metric T_{cl}, the expression T_p describes a concave metric that defines the maximum throughput limited by the characteristics of the network path from server to client: Let $T(n_i, n_j)$ be a metric describing the achievable throughput between two network nodes n_i and n_j and let $p(n_1, n_2, ..., n_m)$ be the path between server node n_1 and client node n_m. Then T_p can be expressed as

$$T_p = \min\left(T(n_1, n_2), T(n_2, n_3), ..., T(n_{m-1}, n_m)\right)$$

Similarly, the expression T_{sp} describes the achievable throughput on the path *sp* which is the subset of p that the client *cl* shares with $N_{sp} - 1$ other clients. It can clearly be seen that T_{cl} decreases with an increasing N_{sp}.

However, when using *multicast* data transmission, the maximum client-observable throughput becomes independent of the number of clients that share the same path. The metric then evaluates to a rather simple

$$T_{cl} = T_p \tag{2}$$

showing that the maximum throughput observable by *cl* is now independent of the number of other clients it shares the network path to the server with.

Multicast data transmission can also be beneficial for the **latency** of communication taking place, because it can cut down on server answer time. First, for the *unicast* case, the delay or latency observed by a client *cl* can be defined as

$$L_{cl} = L_p + t_{srv} * (N_{sp} - 1) \tag{3}$$

Within L_{cl}, the expression L_p describes an additive metric defining the latency of the client's connection to the server: Let $L(n_i, n_j)$ be a metric that describes the latency

between two network nodes n_i and n_j and let $p(n_1, n_2, ..., n_m)$ be the path between server node n_1 and client node n_m. Then L_p can be expressed as

$$L_p = L(n_1, n_2) + L(n_2, n_3) + ... + L(n_{m-1}, n_m)$$

The term t_{srv} in L_{cl} describes the time the server needs to serve a single client. This includes preparing data, making calculations and transmitting data. N_{sp} is defined as in the throughput metric above. It can be seen that given a non-zero t_{srv}, L_{cl} increases with an increasing N_{sp}. The higher t_{srv}, the stronger the effect.

The benefit of multicast data transmission is that it eliminates the possible delay a client might encounter while waiting for others to be served: Because data is now sent only once instead of N_{sp} times, the latency observed by *cl* when using multicast data transmission is described by

$$L_{cl} = L_p \tag{4}$$

Related to Security. The first aspect of security, access control, is equally important in all considered use cases: may it be in presentations, training courses or professional collaboration meetings, the fundamental fact is that there is a desktop being shared. Naturally not everybody, not even in the local subnet, should be granted access without basic authentication. Thus, the system should provide at least a simple **access control** feature to lock out unwanted users. Additionally, some form of tiered access control like allowing full or view-only access may be useful for the considered use cases as well.

The other aspect of security is confidentiality, availability and integrity of the communication that is taking place. This may not be much of an issue if data is sent and received only in a private, secured local area network, but it definitely becomes an issue when geographically remote users take part. Especially when data is transmitted over untrusted networks like the Internet, it has to be **encrypted** somehow to prevent eavesdropping or spoofing.

Related to Usability. By having looked closely at the considered use cases, it became clear that different people with different levels of technical knowledge would use the system. Therefore the system should feature an intuitive and easy to use graphical user interface at the client side that provides users with all fundamental operations like connecting, interaction and disconnecting. The GUI should preferably adhere to the user interface conventions of the OS it is running on.

Furthermore, it was concluded that one the most significant issues hampering usability is unnecessary and recurring configuration. For instance, requiring the user to know of IP addresses and ports in order to connect to some service is considered harmful. The user application should rather support mechanisms like **automatic service discovery**, simply providing the user with a list to choose from. Preferably, users should be able

to connect with a single click instead of having to care about technical details like IP addresses or ports.

Related to Portability. As stated in subsection 2.2.1, for the electronic classroom use case (for instance schools), platform independence may not be overly important since it can be expected that a homogeneous supply of computers is available, but especially for the presentation and professional collaboration use cases it is very likely that participants run different operating systems on their client computers. Thus the used software on the client side should be **cross-platform** or at least available for Unix, Mac OS X and Windows.

2.2.3 Summary

In conclusion, the requirements posed by the considered real-time collaboration use cases can be summarised like in figure 6: this diagram shows identified functional and non-functional requirements and their inter-relationship regarding as to which functional requirements help to satisfy which non-functional ones as described above.

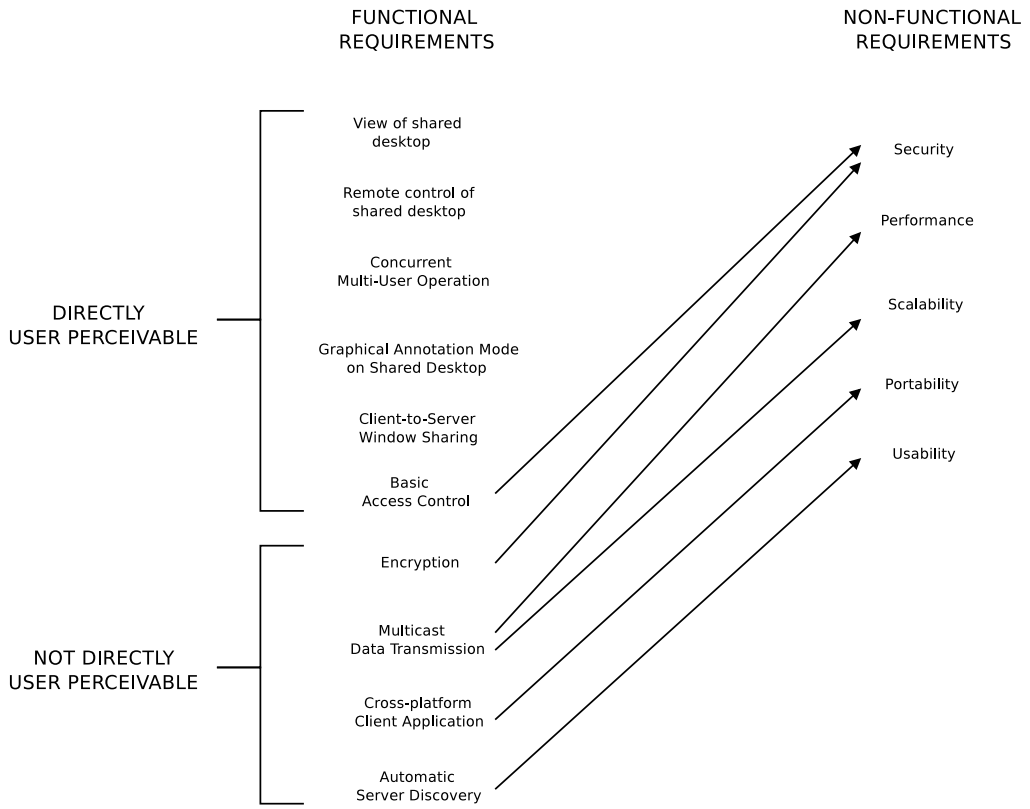


Figure 6: Summary of functional requirements and their relation to non-functional ones.

3 State of the Art

3.1 Basic Principles regarding Real-Time Collaboration Systems

3.1.1 Classification

Collaboration, when denoting working together with the help of computers, is commonly referred to as *computer supported cooperative work* (CSCW) in literature. CSCW can be defined as »*computer assisted coordinated activity such as communication and problem solving carried out by a group of collaborating individuals*« [78, p. 1], but the term is also used to name the multi-disciplinary field of research that deals with understanding of social processes and the design, implementation and evaluation of technical systems supporting social interaction [89].

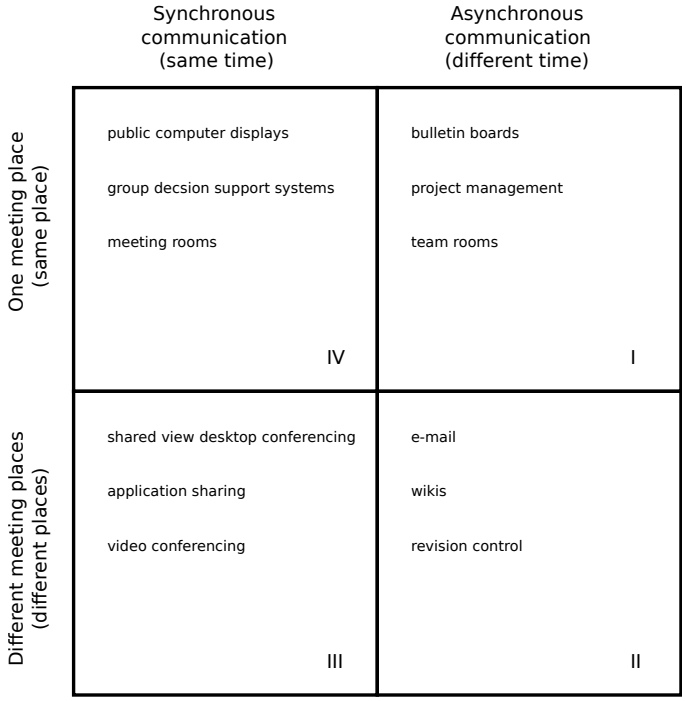


Figure 7: Classification of groupware by space and time.

Such multi-user software systems facilitating CSCW are commonly referred to as *groupware* [78], but this term is sometimes also used to describe the software together with the social group processes [99]. Since the aspect of social processes is contained within the term CSCW, this work uses the former definition and refers to groupware as the software system supporting CSCW. The are various possible classifications of groupware. The most common one is the space-time taxonomy [98] that classifies collaboration systems using this two aspects as in figure 7. Other possible classifications use relations between

persons and artefacts [84], the interdependencies of communication, coordination, cooperation [120] or different application classes [86] to categorize collaboration systems. This work uses the space-time taxonomy and focuses on real-time collaboration systems that facilitate desktop conferencing and application sharing (quadrant III in figure 7).

Social Entities

An important insight into the nature of collaboration systems is that they are technical systems that are tightly interwoven with social systems. Therefore they can also be referred to as socio-technical systems. The social component is characterised by different forms of interaction between different social entities.

A possible classification of social entities distinguishes between dyads, groups, teams, social networks, communities and organisations [89, p. 16 ff]: Dyads are social entities that consist of exactly two persons, groups comprise more people. What characterises dyads and groups is their differentiation against other people, which can take the form of inward (e.g. through self-identification) or outward (e.g. through official club membership) differentiation. Teams are defined as groups that pursue a target. Social networks, on the other hand, are characterised by the social relations of the entities they consist of. Communities, in contrast to social networks, are defined as groups of people sharing a common culture. Like teams, communities also pursue a target, but they have a bigger number of members that do not necessarily know each other. Finally, organisations are seen as social entities that pursue a target with the help of social structuring and coordination.

All these categorisations of social entities are not disjunct. However, it can be stated that the real-time collaboration use cases considered in section 2 do not comprise all of them but only dyads, groups and teams. Real-time interaction of a larger group of people would be too noisy.

Forms of Social Interaction

Concerning forms of interaction between the mentioned social entities, communication, consensus building, coordination, awareness and cooperation can be listed as the most fundamental ways of interacting [89, p. 8]. Real-time collaboration systems perform particularly well with respect to communication, awareness and cooperation. Consensus building and coordination are less of a focal point.

Precisely because they allow users to concurrently interact, real-time collaboration systems do not necessarily have to provide special means to facilitate *consensus building* and *coordination*: users are able to do this by communicating in real-time. They can concurrently ask and answer questions, very much like in a real-world face-to-face setting. Synchronous *communication* furthermore allows for some imprecision when starting to convey ideas, the correct meaning can be worked out by asking and answering questions. This way, ideas can be communicated more rapidly than with asynchronous communication where thoughts have to formulated as precisely as possible from the start on because further clarification would take a lot of time.

The synchronous nature of real-time collaboration systems is also the reason they facilitate mutual *awareness* of users: When users communicate in real-time, they are necessarily aware of each other.

Finally, this mutual awareness and the ability to communicate are prerequisites for successful *cooperation*. Here support for joint handling of electronic artefacts is of great importance since cooperation almost always involves working with shared electronic documents or data.

3.1.2 Common Technical Realisation

Figure 7 on page 14 lists shared view desktop conferencing, application sharing and video conferencing systems as examples for different-place real-time collaboration systems. Such systems are often realised by integrating different technologies [89, p. 131].

When looking at such systems from a multi-user-support perspective, they can be categorized in two classes, like already pointed out in section 1.1: One the one hand, there are special multi-user tools that support concurrent multi-user interaction but are confined to a single application. On the other hand, there are more general systems that allow sharing any kind of desktop application but are limited in terms of concurrent multi-user support.

The first group according to this classification are real-time collaboration systems that provide a single application which is specifically designed for concurrent multi-user support: This includes collaborative text editors such as SubEthaEdit [45], Gobby [13], SynchroEdit [47] or EtherPad [10] as well as shared whiteboards and drawing applications such as Scriblink [39], Twiddla [55], iScribble [19] or PaintChat [35]. Most of the text editors are implemented as native applications, with the exception of EtherPad, which is a web application implemented in JavaScript. The mentioned shared whiteboards and drawing applications are all web-based and implemented using Java or Adobe Flash.

The other group of real-time collaboration systems according to the multi-user support classification are shared view desktop conferencing systems that allow users to operate any kind of standard desktop application. Some of them support sharing single applications instead of whole desktops as well. Commonly, such systems are based on a remote desktop technology such as the X Window System, VNC or RDP. Some are also realised using other, proprietary protocols or are built on web application technologies like Java or Flash.

Since one of the focal points of this work was to create a real-time collaboration system that allows its users to *concurrently* interact with *any kind* of application on a *standard desktop*, this second group, comprised of shared view desktop conferencing systems, is examined more closely in section 3.2.

3.2 Survey of Existing Real-Time Collaboration Systems

After identifying the requirements posed by the different considered use cases, the next step was to do a survey of related work to see what had already been done and what basic technology and tools were available to build upon.

A first obvious candidate was the *X* Window System version *11* that by design supports transmission of graphical user interfaces over a network. A promising alternative was found to be the desktop sharing system *V*irtual *N*etwork *C*omputing whose most notable feature is its simplicity and thus wide distribution among operating systems and devices. Software based on the *R*emote *D*esktop *P*rotocol developed by Microsoft was also considered.

Thus, results of the investigation are split up into four subsections, the first three ones containing results belonging to the three major remote desktop technologies X11, VNC and RDP. The last subsection contains software whose underlying protocol is different from these major three or is unknown.

In each subsection, the individual works were examined with regard to their conformance to the most fundamental functional requirements identified in section 2.2. Additionally, all examined software products were checked for source code availability, which is a fundamental requirement in case the system in question would be chosen to be extended.

Results are summed up in a table and a short textual roundup at the end of each subsection.

3.2.1 Based on the X Window System

The X Window System, also just called X or X11, is a windowing system *and* network protocol. It is the most commonly used software to display a GUI on Unix-like operating systems. Since the X Window System was designed from the ground up with network transparency in mind, all X-based applications can be displayed and controlled remotely or locally. The main catch here is that bare X11 only supports one receiver for each application instance: This way, applications cannot be shared. There are, however, some X11-based tools that facilitate sharing of applications between several participants.

x2x

One basic way to control an existing X session is the tool x2x [68]. With x2x, keyboard and mouse of a local X display are able to control a another remote X display. Depending on its configuration, x2x creates an invisible, one pixel wide window at one edge of the local screen. If the mouse pointer moves over this window, x2x sends mouse and keyboard commands to the remotely controlled computer. If the cursor moves back, the desktop of the local machine is controlled as before. x2x thus is suitable for use as a basic remote control, but does only support the X Window System, does not support transmission of display contents and has no collaboration support whatsoever.

xtv

With xtv [72], it is possible to locally *view* a remote X display in a window. xtv does not allow the user to control the remote display and is tied to the X Window System. Because it is transferring the entire screen content uncompressed via unicast it is quite slow and barely usable in a wireless environment.

NoMachine NX

With NoMachine NX [33] a remote X display can be controlled through a window appearing on the local machine. NX client implementations exist for most operating systems, users are not tied to operating systems that have a X Window implementation.

NX acts as a proxy between the remote controlled X display and the client: the NX proxy compresses the data flow and creates a cache of already transmitted data (e. g. icons). This eliminates many unnecessary round-trips between X client and X server. The NX proxy architecture also makes an X connection somewhat stateless: If the network connection terminates unexpectedly while running a traditional X session, all applications terminate as well because state is stored at the server *and* the client. If it is only the client's connection to the NX proxy that terminates, the applications keep on running, the client can eventually reconnect and continue working where it stopped.

While the underlying libraries are open source, the client and server applications are not. While there is a free implementation of the server called FreeNX [11], it seems it is not very well maintained and could not be made to run. The proprietary server only supports three registered clients, on the other hand its performance is relatively good: It is possible to play videos at a viewable frame rate over an 11 Mbps connection. However, only the proprietary client application exists, there are no free client implementations. Furthermore, NX makes no use of multicast, all display contents are transferred unicast.

NX does not support collaborative features like annotations, application sharing or multi-user operation of the remote display.

XMX

XMX [70] is a »X Protocol Multiplexer«: it claims to allow an X session to be distributed among several other X displays, which can then view and control this session. Like NX, XMX acts as a proxy between communication partners, the remote X session is displayed in a local window. What sets XMX apart from NX is that it incorporates some multi-user features like a basic floor control. It does not, however, support other collaboration features like annotations or window sharing.

Unfortunately the last release of XMX was made in 1999, so this software is rather outdated. It was impossible to obtain or produce a version that would run on today's X Window System implementations.

MPX

While not exactly being a remote display technology in itself, MPX [34] is a very useful extension of the X Window system in terms of multi user collaboration support. MPX

stands for Multi Pointer X and allows fully concurrent multi-user operation with several pointers and keyboards at the window system level. According to [96], MPX is the first incarnation of a real Groupware Windowing System: Instead of modifying existing applications, multi user support is built into the underlying windowing system. This way multiple users can simultaneously interact with *different* applications on the same display. To allow control of a *single* application by several input devices at the same time, an application has to be modified to be made multi-device aware.

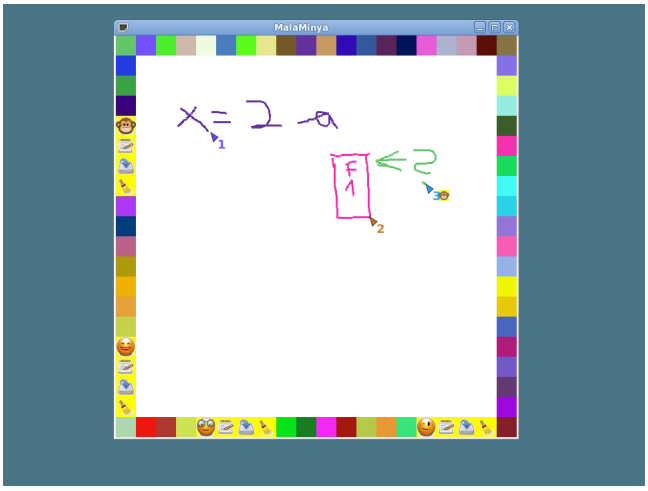


Figure 8: Several MPX pointers operating a shared scribble sheet.

There is an experimental multi-pointer window manager available [95] that demonstrates MPX features like simultaneous moving or resizing of windows with two or more mouse pointers.

Comparison

Of the five examined X11 solutions, only NoMachine NX and XMX provide full remote desktop access including view *and* control. Multi-user operation is only provided by XMX, but it just allows turn-taking. Fully concurrent multi-user operation is only provided by MPX, but on the other hand, this system provides no remote desktop functionality at all and thus has to rely on other tools to provide remote desktop functionality. Annotations, client-to-server window sharing, multicast or server discovery are provided by neither of the tools examined, but they all have source code available, making extensions possible.

Table 1: Comparison of X Window System Software.

	View	Control	Multi-User Operation	Annotations	Client Window Sharing	Multicast	Autom. Server Discovery	Cross-Platform Client	Source Code Available
x2x	-	✓	-	-	-	-	-	-	✓
xtv	✓	-	-	-	-	-	-	-	✓
NoMachine NX	✓	✓		-	-	-	-	✓	(✓)[1]
XMX	✓	✓	(✓)[2]	-	-	-	-	-	✓
MPX[3]	-	-	✓	-	-	-	-	-	✓

3.2.2 Based on VNC

VNC or Virtual Network Computing is a remote desktop technology that tries to adhere to the *thin client* paradigm: It tries to keep the client as simple as possible and concentrates most of the complexity at the server side. Therefore, the underlying protocol called RFB (for Remote Framebuffer Protocol [115]) is intentionally kept as simple as possible to ease client side implementation. VNC simply transmits (optionally compressed) image data to the client which in turn sends back mouse and keyboard commands to the server. The image data sent by the server can be encoded in different ways, which are negotiated by server and client at session start-up. In contrast to the X Window System, a VNC connection is a *stateless* one: all state is stored at the server so that when a connection dies unexpectedly, the application keeps on running on the server. The client can simply reconnect and continue working where it stopped.

Because of the simplicity of the RFB protocol, client implementations are really widespread and do exist for almost all major and minor operating systems [42].

RealVNC

RealVNC [37] is the direct descendant of the original VNC software suite, developed by the same team that created the first implementation of VNC. RealVNC is available in

[1] NX libraries are open source, but server and client applications are proprietary.
[2] XMX supports basic floor control, but no real concurrent multi user operation.
[3] MPX is not exactly a remote desktop technology, but a viable extension of X11 because it allows fully concurrent multi-user operation, so it is included here as well.

three different flavours: a free, open source edition with the basic VNC features (available for Unix); a personal, commercial edition featuring text chat, printing support and encryption (available for Windows); and an enterprise edition with enhanced authentication and encryption (available for Unix, Windows and Mac OS X). Probably because it is an offspring of the original VNC software, RealVNC supports only the original VNC encodings, although more performant ones have been developed by other projects [56, 53]. Besides text chat support, RealVNC does neither offer any multi-user support nor multicast.

UltraVNC

UltraVNC [56] is an open source Windows implementation of a VNC server and client. It features its own »ultra« compression scheme and a so called »mirror video driver« that allows the server application to get notified about screen updates without constantly polling the framebuffer, resulting in lower CPU load. The bundled UltraVNC client application is able to view and control any VNC server and additionally provides basic file transfer functionality when connected to an UltraVNC server. Besides these features, server and client do not provide any multi-user collaboration features or multicast data transfer capabilities.

TightVNC

Unlike UltraVNC, TightVNC [53] is available for both Microsoft Windows and Unix-like operating systems. TightVNC was the first implementation to support the JPEG-based »tight« encoding for image data, hence the name. This encoding is a lossy one and achieves very good compression ratios compared to lossless encodings. The Windows version of the server is able to share the whole desktop or just a single window. The TightVNC client application is able to connect to any VNC server, but only when connected to tight-enabled servers it is able to use this lossy encoding. TightVNC does neither support any multi-user features nor multicast.

xf4vnc

Xf4vnc [69] is a VNC server that is – like the Unix variants of RealVNC and TightVNC – also an X server. This mode of operation means that a new X11 session that is exported via VNC is spawned for every client. Like RealVNC and TightVNC, xf4vnc is not able to share an existing session this way. However, it does allow sharing of OpenGL applications, although not hardware accelerated (RealVNC and TightVNC provide no support for OpenGL applications at all). Xf4vnc also supports redirecting the OpenGL command stream to clients to be rendered there, as opposed to rendering the image on the server. Clients have to have support for this system called Chromium [7], though.

Additionally, xf4vnc also provides an X server plug-in module that is able to share an *existing* X11 session. However, this requires changing the X server configuration in superuser mode. Furthermore, the module does not work with recent X server implementations.

Both implementations of xf4vnc do not provide any multicast data delivery or multi-user collaboration features.

x11vnc

X11vnc [116] is an open-source VNC server for Unix-like operating systems. Unlike traditional VNC servers for Unix systems, which create a new session for every client, it allows sharing of an existing user session by connecting to an already running X server. This way x11vnc acts as a client of the running X server whose display it is exporting and as a server to VNC clients which receive that display. Because it basically works like a sophisticated screen scraper, x11vnc is also able to export accelerated OpenGL applications, a thing that traditional Unix VNC servers are not able to do because they are both X11 and VNC server in one application and lack the proper X11 OpenGL extensions[4]. X11vnc (through the LibVNCServer library [21]) furthermore supports all major VNC encodings including »tight« and »ultra«, can do server-side screen scaling, is able to share the whole desktop or just single windows and has support for automatic server discovery via Zeroconf [82]. It does not have any multi-user features or support for multicast data transfer, though.

Vino

Vino [58] is the standard VNC server of the GNOME desktop environment. It works exactly like x11vnc in that it is able to share an existing session (possibly with OpenGL applications), with the exception that Vino can only share the whole desktop, not single applications. It also provides Zeroconf service advertisement and supports the same VNC encodings as x11vnc (by using LibVNCServer [21]), but unlike x11vnc, it is very tightly integrated into the GNOME environment.

Apple Remote Desktop

Apple Remote Desktop [4] is the default remote administration software suite used in Mac OS X that uses VNC for graphical remote desktop access. It also provides other features like distribution of software packages, remote batch jobs and remote monitoring. What is missing though is multi-user support or multicast data delivery.

Collaborative VNC

Collaborative VNC [8] is a patch to TightVNC 1.2.9 Unix version that extends the original software with some multi-user support: Each new client gets a uniquely coloured and labelled mouse cursor that is drawn into the VNC server's framebuffer. This has the advantage that it in principle works everywhere but has the drawback that these mouse cursors are only known to the VNC server and its clients. The VNC server's host operating system does not know about these multiple cursors. Collaborative VNC therefore implements a simple floor control mechanism that maps several client mouse pointers

[4] RealVNC and TightVNC provide no X11 OpenGL (GLX) extensions at all, xf4vnc does, but is not using hardware acceleration.

onto one. This way, it is not possible to interact simultaneously with applications on the server desktop. Furthermore, Collaborative VNC uses a modified RFB protocol, it is not compatible with existing VNC viewers.

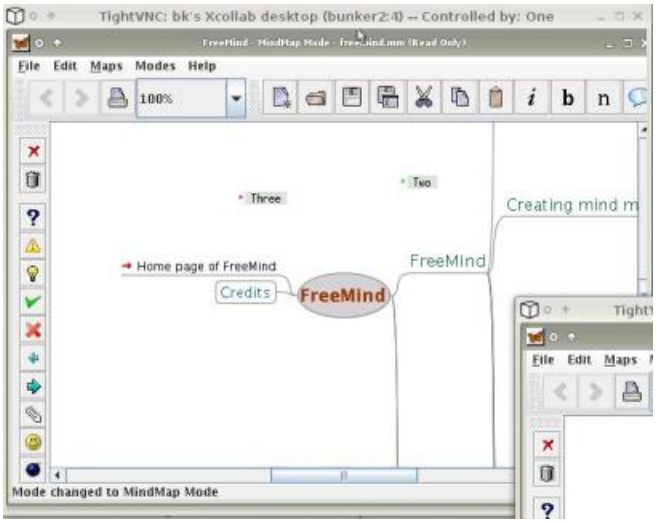


Figure 9: Collaborative VNC showing two distinct client cursors.

DrawTop

The DrawTop project developed by the University of Sydney[119, 9] works similar to Collaborative VNC in that it draws into the VNC framebuffer. However, DrawTop is not a modification of an existing server, but sits as a proxy between VNC server and clients and works with standard VNC clients. It offers a transparent framebuffer overlay that clients can draw into, making annotations on the original server desktop. Also, one client at a time can take control of the underlying desktop. It is not possible to have multiple clients control the server's desktop. DrawTop uses the same VNC library as x11vnc and Vino do [21], so supports the same encodings.

SharedAppVNC

SharedAppVNC [40] is a tool for remote collaboration that allows its users to share individual applications between them. SharedAppVNC does not make a strict distinction between client and server: every participant can act as a server, sharing applications, and as a client, receiving applications. Shared applications can either be set to »view only« or can be controlled by the receivers, where every application appears in its own movable and resizable frame. The software is available for Linux, Mac OS X and Windows, but uses a modified RFB protocol, so it is not compatible with existing VNC servers or clients.

MetaVNC

Similar to SharedAppVNC, MetaVNC [28] is a window aware VNC server and client, but it uses a different approach from the user perspective. First, the MetaVNC server always shares all windows, users are not able to select specific ones. Second, the MetaVNC viewer does not place each received window into its own manageable frame, but instead uses a single fullscreen window with a transparent background to draw received windows into. It is based on TightVNC 1.3.9, so otherwise supports the same features as TightVNC.

TightProjector

TightProjector [52] is a commercial VNC server available for Windows that sends all data via multicast. A special client application is needed to receive this multicasted VNC data. Users are only able to view the server's desktop, it is not possible to control the remote side. Analysis of network traffic revealed that TightProjector seems to do no error recovery whatsoever, it simply sends fullscreen updates at regular time intervals.

MulticastVNC

MulticastVNC [31] is a discontinued Java VNC proxy that features transmission of VNC data via multicast. The MulticastVNC proxy connects to a VNC server as a normal client and multicasts the data it gets from the server. A modified Java VNC viewer is used to receive the multicasted VNC data. Probably because the software was developed for tele-teaching, it only possible to *view* the remote desktop, multicast clients are not able to *control* the remote desktop. MulticastVNC does not do any multicast error recovery and only supports the Hextile [115] encoding which does not compress data very well compared to »tight« or »ultra« encoding.

TeleTeachingTool

TeleTeachingTool [51] is an extension of the original MulticastVNC into a feature-rich software suite for distance learning. The main application differentiates between two fundamental modes of use: in »lecturer« mode, the actions on the user's desktop are recorded and sent to connected clients. These run the software in »student« mode. Only the lecturer can control the desktop or make annotations, clients are just able to view the lecturer's desktop. Because it builds on MulticastVNC, TeleTeachingTool's multicast mode also does no error discovery and provides no way to deal with lost packets – it simply sends a fullscreen update at regular time intervals [124, p. 6]. There is also a new implementation that does not use VNC anymore but relies on RTP for multicasting a video stream [50].

Win2VNC / x2vnc

Win2VNC [59] and its Unix counterpart x2vnc [94] are VNC clients that offer a unique mode of control: the main principle is the same as with x2x (section 3.2.1 on page 17): They create an invisible, one pixel wide window at one edge of the local screen. If the

mouse pointer moves over this window, they send mouse and keyboard input to the remote display. When the cursor is moved back, the desktop of the local machine is controlled as before. While this is definitely usable if the user user is able to physically see the remote display (like in a multi-screen setup), this way of remote controlling obviously is quite useless if the remote machine is located somewhere else. Both tools provide remote *control* and nothing more, but are listed here for completeness' sake and because they provide an interesting way of interacting with a VNC server.

Comparison

The majority of the VNC products looked at provide full remote desktop access with view and control functionality. The exception are the tools that have some kind of multicast support: these are all view-only. Only two solutions, Collaborative VNC and DrawTop, feature multi-user operation, but again only in a sequential fashion without fully concurrent interaction. DrawTop also is the only VNC software to fully support on-screen annotations, TeleTeachingTool only provides annotation facilities for the user locally operating the shared desktop. Furthermore only a single system, SharedAppVNC, provides client-to-server window sharing. None of the examined software products meet all of the posed functional requirements, though most of them are open-source, so could possibly be extended.

Table 2: Comparison of VNC Software.

	View	Control	Multi-User Operation	Annotations	Client Window Sharing	Multicast	Autom. Server Discovery	Cross-Platform Client	Source Code Available
Win2vnc / x2vnc	-	✓	-	-	-	-	-	-	✓
TightProjector	✓	-	-	-	-	✓	-	-	-
MulticastVNC	✓	-	-	-	-	✓	-	✓	✓
TeleTeachingTool (old)	✓	-	-	(✓)[5]	-	✓	-	✓	✓
RealVNC	✓	✓	-	-	-	-	-	✓	✓
UltraVNC	✓	✓	-	-	-	-	-	✓	✓
TightVNC	✓	✓	-	-	-	-	-	✓	✓
xf4vnc	✓	✓	-	-	-	-	-	✓	✓

[5] TeleTeachingTool offers annotations only for the lecturer, not for connected clients.

	View	Control	Multi-User Operation	Annotations	Client Window Sharing	Multicast	Autom. Server Discovery	Cross-Platform Client	Source Code Available
MetaVNC	✓	✓	-	-	-	-	-	✓	✓
Apple Remote Desktop	✓	✓	-	-	-	-	✓	✓	-
Vino	✓	✓	-	-	-	-	✓	✓	✓
x11vnc	✓	✓	-	-	-	-	✓	✓	✓
Collaborative VNC	✓	✓	(✓)[6]	-	-	-	-	✓	✓
SharedAppVNC	✓	✓	-	-	✓	-	-	✓	✓
DrawTop	✓	✓	(✓)[7]	✓	-	-	-	✓	✓

3.2.3 Based on RDP

RDP or Remote Desktop Protocol [107] is a Microsoft-developed extension of the ITU-T T.128 application sharing protocol [97] and is – like VNC – used to provide remote users with a graphical user interface transmitted over a network. In addition to transmitting the GUI of an application, RDP also allows sounds to be redirected to a remote machine. RDP is used as the fundamental remote desktop technology in almost all versions of Microsoft Windows. There are clients for some other operating systems as well, although it seems RDP clients are not as widespread as VNC clients. RDP servers mostly exist for Windows operating systems, although there is one implementation [71] for Unix-like systems, too.

Windows Remote Desktop Services

Windows Remote Desktop Services [65] is what comes by default with most Windows versions. Depending on the version of Microsoft Windows used, Remote Desktop Services simply make the entire desktop of a Windows machine available or are able to share single applications as well. Using an RDP client application, it is possible to view and control a remote desktop or several single applications, but there is no multi-user support on the server side nor on the client side: Remote Desktop Services do neither offer concurrent

[6] Collaborative VNC supports basic floor control, but no real concurrent multi user control.
[7] DrawTop supports basic floor control, but no real concurrent multi user control.

multi-user operation nor are *clients* able to share applications back to the server. All communication is done unicast.

Windows Desktop Sharing

Windows Desktop Sharing [60] is available from Windows Vista onwards and, instead of creating a new session for every user like Windows Remote Desktop Services, is able to share the existing session of a local user. Otherwise, it provides exactly the same functionality as Windows Remote Desktop Services do.

Windows Meeting Space

Windows Meeting Space [61] is remote desktop software that comes with Windows Vista and allows users to share their Vista desktop to up to nine other Vista users by sending invitations to them. Windows Meeting Space also supports automatic server discovery, file transfers and even automatic setup of an ad-hoc wireless network if no suitable network is found. However, Windows Meeting Space is only available for Windows Vista and lacks features like concurrent multi-user operation or multicast data transfer.

xrdp and rdesktop

Xrdp [71] and rdesktop are the only available open source implementations of a RDP server and RDP client. The xrdp server application is available for most Unix-like operating systems; however, it does only support a subset of RDP. Specifically, the ability to redirect sound or share single applications is missing. Interestingly, xrdp talks RDP with the outside world, but seems to use a VNC server internally. On the other hand rdesktop [36] – as the client application – supports playing of sounds, is ported to most major operating systems and is compatible with Microsoft Windows servers, but does not offer any additional functionality compared to its proprietary counterparts.

Comparison

The examined RDP systems all support viewing and controlling a remote desktop, but none provides collaboration features such as multi-user operation, annotations on the shared desktop or client-to-server window sharing. Also, none of the considered tools can multicast a shared desktop. Unfortunately, only xrdp and rdesktop have their source code available, which makes these tools the only ones possible to extend.

Table 3: Comparison of RDP Software.

	View	Control	Multi-User Operation	Annotations	Client Window Sharing	Multicast	Autom. Server Discovery	Cross-Platform Client	Source Code Available
Windows Remote Desktop Services	✓	✓	-	-	-	-	-	✓	-
Windows Desktop Sharing	✓	✓	-	-	-	-	-	✓	-
Windows Meeting Space	✓	✓	-	-	-	-	✓	-	-
xrdp / rdesktop	✓	✓	-	-	-	-	-	✓	✓

3.2.4 Others

The software listed in this section either uses a protocol different from X11, VNC and RDP or it is simply not known what the underlying protocol is. The latter is the case with many proprietary commercial products.

Synergy

Synergy [48] works very much like x2vnc or Win2VNC (section 3.2.2 on page 24), but instead of RFB, it uses a custom, self-developed protocol. Otherwise, the mode of operation is the same: The user can move the mouse pointer over one edge of the screen and Synergy will send mouse and keyboard input to the remote machine instead. Like x2vnc or Win2VNC, there is no transmission of display data. Synergy is open source and available for Unix, Windows and Mac OS X.

Mikogo

Mikogo [30] is a free to download (but not open source) screen sharing program available for Windows and Mac OS X, with the Windows version being far more advanced feature-wise. It allows presenters to share their screen (or single applications) to others, make annotations and share files. Clients can view what is happening on the presenter's desktop and can optionally take control from the presenter. Clients can also highlight some item on the presenter's screen by clicking on it: a coloured cursor will appear on the presenter's desktop, but the client is not able to control the remote side. It is,

however, possible to switch the presenter role dynamically in a session. To start a new session, a user account at mikogo.com is necessary. Upon session startup, the presenter receives a 9-digit number from the central mikogo server, which serves as an invitation ID for other participants. Besides the features mentioned, Mikogo does neither support multi-user operation nor multicast data transfer.

Yuuguu

Yuuguu [73] is a commercial remote desktop solution available for Linux, Mac OS X and Windows. Basically it is a chat application with built in screen sharing: a presenter can chat with other users and share her screen with them over an encrypted channel. Users running the client application or web browser applet are then able to view the presenter's desktop and can optionally request control. Yuuguu has no support for annotations, multi-user interaction or multicast.

GoToMyPC

GoToMyPC [14] allows Mac OS X and Windows users to share their screen to Unix, Mac and Windows users. It supports encryption, remote printing, file transfer and audio and works through firewalls by using relay servers. The software is available for a 30-day trial, a GoToMyPC account is necessary to run it. GoToMyPC does not support any multi-user features, annotations or multicast.

LogMeIn

LogMeIn [25] offers a whole family of remote desktop products, the LogMeIn Pro^2 being the main remote desktop application. It is available on a yearly subscription basis and supports Mac OS X and Windows. Besides screen sharing, LogMeIn Pro^2 features transmission of audio data, file transfer, remote printing and remote system monitoring. However, there is no annotation, multi-user or multicast support.

Timbuktu Pro

Motorola Timbuktu Pro [54] is available for Mac OS X and Windows. It allows for screen sharing, text and voice chat and file transfer over SSH encrypted communication channels. Multiple clients can connect to one server, but only one at a time can be in control. The client application allows viewing multiple remote displays at once and supports server discovery via Zeroconf [108]. Timbuktu Pro makes use of the Skype API in order to find known participants and to work through firewalls. There is no multicast or advanced multi-user support.

GO-Global

GO-Global by GraphOn [12] is a commercial remote access software that concentrates on serving the GUIs of applications running on a central server to remote users. Received applications are displayed using a special client application or web browser applet. Both

client and server are available for Unix, Mac OS X and Windows. GO-Global supports remote printing and encryption, but has no multi-user support and only uses unicast.

Lotus Sametime Unyte Share

Sametime Unyte Share [26] is a commercial one-on-one screen and document sharing software developed by Lotus. It is only available for Windows. Sametime Unyte Share supports sharing the whole desktop or single applications, file transfer and text chat over an encrypted channel with *one* other participant, who receives an invitation and then runs a web browser applet. Sametime Unyte Share has no support for multiple users.

Symantec PCAnywhere

PCAnywhere by Symantec [46] is a remote control software available for Unix, Mac OS X and Windows. Its main features are remote desktop control, file transfer, text chat, remote printing support, encryption, session recording and firewall traversal using gateway servers. PCAnywhere is one of the few commercial solutions that make use of multicast data transmission: it has a so-called »conference mode« where multiple clients receive the desktop of a server machine via multicast, but only the first client to connect can exercise control [74, p. 88]. There seems to be no multi-user support.

Adobe Connect

Adobe Connect is a web conferencing software that uses an Adobe Flash browser plugin that enables users to chat with each other (optionally with audio and video support), to transfer files, make a vote on a discussion item, use a shared whiteboard and record sessions. The software comes in two flavours: one is called ConnectNow [1], the other ConnectPro [2]. The latter mainly differs from the basic version in that it supports more users and better infrastructure for managing large groups: the browser plugin is supplemented by a web portal where users are able to upload documents for online seminars, create schedules, manage groups and invite others to join. The trial version allows up to three participants, the licensed version several thousands.

The use of Flash makes the solution somewhat cross-platform, although the most prominent feature – screen sharing – does not seem to be enabled on Unix platforms. However, Unix users are able to receive entire desktops or single applications shared by Windows users. It is also possible to draw annotations on shared windows, but only for the presenter. Every user can request control of shared windows from the presenter, but there is no support for concurrent multi-user interaction.

Microsoft SharedView

SharedView developed by Microsoft [29] offers sharing of the whole desktop or applications, text chat and file uploading for up to 15 participants running Microsoft Windows. A Windows Live ID is required to start a new session, others can then join in after having received an invitation. As an optional feature, each user gets a personal coloured mouse cursor that enables her to mark up regions of the screen. Additionally, every user

can request control from the window sharer, but only one user at a time can exercise control.

Windows Multipoint

Microsoft Windows Multipoint is an umbrella term for a multi-pointer software development kit [63] and a set of associated applications developed using this SDK. The framework, available for Windows 7, Vista and XP, allows to create applications that support up to 25 independent mouse cursors. It has no notion of multiple keyboards, though.

Applications making use of the multipoint SDK are Windows Multipoint Server [64] and Microsoft Mouse Mischief [62]. The former is a multiseat solution that allows multiple users to work at one computer, using several monitors, mice and keyboards, all *directly* connected to the server. Normally Windows Multipoint Server provides each user with their own session, but it is also possible to share one session and work together on one desktop using multiple mouse cursors. The second application, Mouse Mischief, is a plugin for Microsoft PowerPoint that extends this presentation software with support for multiple pointers. There is no mention of support for multiple keyboards.

SPICE

SPICE [43] is both a protocol and software collection to view and control virtualized desktop machines over a network. Unlike VNC, and similar to X11 or RDP, SPICE submits 2D drawing commands instead of image raster data. In addition, it heuristically detects video playback and streams videos as MPEG. SPICE also provides remote audio playback and capture as well as synchronization of audio and video. On the other hand, it has no multi-user support nor the ability to multicast data. Open source SPICE implementations are available for Unix and Windows.

TeamViewer

TeamViewer [49] is a proprietary cross-platform remote desktop software available for Linux, Mac OS X and Windows. It allows a users to share their desktops or single applications[8] to another user or to receive the other user's desktop. It is possible to switch directions on the fly, but only either receiving windows or sharing windows is possible at a time, not both. TeamViewer furthermore features text, audio and video chat, session recording, file transfer, setup of a virtual private network, a web browser applet, firewall traversal, and encryption of communication. With TeamViewer it is possible to make graphical annotations on the desktop, but only for the sharer, not the receiver.

TeleTeachingTool

As mentioned in section 3.2.2 on page 24, there also is a new implementation of TeleTeachingTool [50] that uses RTP instead of VNC for multicasting a video stream of the lec-

[8] This is not possible with the Linux version.

turer's desktop, so it is listed here. Like the original TeleTeachingTool, the new implementation differentiates between two fundamental modes of use: in »lecturer« mode, the actions on the user's desktop are recorded and sent to connected clients who run the software in »student« mode. Only the lecturer can control the desktop or make annotations, clients can only view the lecturer's desktop.

THINC

According to its authors, THINC is »a virtual display architecture for thin-client computing« [79]. The THINC architecture comprises an interface to the server's graphics hardware at the device driver level, a custom network protocol and server and client applications. Because it has device driver level access to the server's display, THINC does not have to poll the display constantly, but is immediately notified when an update occurs, making the display to server communication very efficient[9]. Another benefit of this approach is that the server is able to detect videos being played back and can encode them specifically. The network communication protocol used by THINC is similar to RFB in that it solely uses simple drawing primitives like »fill region« or »copy bitmap« instead of high-level 2D drawing commands. THINC furthermore supports transmission of audio data and server-side screen scaling (for example for handheld devices). It has no support for multicasting of display data and no notion of multiple users controlling one server, although a multi-user extension [83] exists that adds basic floor control features to THINC. This extension does not allow multiple clients to interact *concurrently*, though.

VNCast

Despite its name, VNCast [109] is actually an RTP multicaster that connects to a VNC server as a normal VNC client and multicasts the image data it receives as a video stream to clients running a custom RTP receiver application. Designed this way, VNCast only allows *viewing* the remote desktop. To exercise control, a traditional VNC client application is still needed.

MAST

The acronym MAST stands for Multicast Application Sharing Tool [91]. As the name implies, MAST was designed to share single applications via multicast. Each user running the (Windows or Linux) application is provided with a list of other currently active participants and can choose which applications to share with them. Unlike THINC, MAST continuously polls the presenter's display for changes. By analyzing the source code [27], it appears that users are only able to view shared applications, but cannot remote control them. MAST can transmit data via unicast or multicast, but while multicasting, it has no knowledge of packet loss whatsoever, instead it resends parts of the screen at configurable regular intervals.

[9] Some VNC implementations [56, 37, 53] use a similar mirror driver approach.

BASS

BASS is an application sharing system developed at University of Columbia [81]. Very much like THINC, BASS interfaces with the exported display at the device driver level. This server component, that allows sharing applications, is only implemented for Windows; client applications are available for Unix, Mac OS X and Windows. Since the BASS server uses a mirror driver to interface with the display hardware, it can be efficiently notified when there are updates and it can also detect videos being played back, which it encodes in a format especially suited for video transmissions. BASS uses a custom network protocol similar to RFB, but encapsulated in the Realtime Transport Protocol RTP to share applications via multicast. It is the only remote application sharing system found that uses multicast data transmission *and* handles lost packets. BASS uses a NACK [76] approach: it discovers lost UDP packets based on sequence numbers and asks the server to retransmit lost ones. Since it employs features provided by RTP and RTCP, BASS additionally has multicast congestion control. Finally, BASS also supports a simple floor control mechanism to let multiple users control one application, concurrent user control is not possible, though. BASS binaries can be downloaded [80], but could not be made to work.

Comparison

Out of the multitude of examined other remote desktop tools, most support full access with viewing *and* controlling of a remote desktop. On the other hand, regarding multi-user operation support is much more scarce: Only three solutions, SharedView, THINC and BASS, feature multi-user operation, though only in the form of turn-taking. The only solution that supports concurrent multi-user operation is Windows Multipoint which otherwise does not feature any remote desktop functionality, much like MPX. Drawing annotations onto the shared desktop is supported by five of the examined tools, though with limitations in three cases. Regarding the third important functional requirement for collaboration, client-to-server window sharing, it can be stated that four solutions support it, although Mikogo and Adobe Connect do have some limitations in this respect. Out of the relatively few tools that are able to distribute a shared desktop via multicast, only BASS supports remote *control*, all others are view-only. Finally, most of the examined software is commercial, meaning no source code is available. That leaves seven tools which can be extended with missing functionality: none of the examined solutions meets all of the requirements posed.

Table 4: Comparison of other Remote Desktop Software.

	View	Control	Multi-User Operation	Annotations	Client Window Sharing	Multicast	Autom. Server Discovery	Cross-Platform Client	Source Code Available
LogMeIn Pro[2]	✓	✓	-	-	-	-	-	-	-
Windows Multipoint [10]	-	-	✓	-	-	-	-	-	(✓)[11]
Synergy	-	✓	-	-	-	-	-	✓	✓
Yuuguu	✓	✓	-	-	-	-	-	✓	-
Unyte Share	✓	✓	-	-	-	-	-	✓	-
GotoMyPC	✓	✓	-	-	-	-	-	✓	-
Timbuktu Pro	✓	✓	-	-	-	-	✓	-	-
GO-Global	✓	✓	-	-	-	-	-	✓	-
SPICE	✓	✓	-	-	-	-	-	-	✓
Mikogo	✓	✓	-	✓	(✓)[12]	-	-	-	-
TeamViewer	✓	✓	-	(✓)[13]	-	-	-	✓	-
VNCast	✓	-	-	-	-	✓	-	✓	✓
Adobe Connect	✓	✓	-	(✓)[14]	(✓)[15]	-	-	✓	-
Microsoft SharedView	✓	✓	(✓)[16]	✓	✓	-	-	-	-
Symantec PCAnywhere	✓	✓	-	-	-	(✓)[17]	✓	✓	-
TeleTeachingTool (new)	✓	-	-	(✓)[18]	-	✓	-	✓	✓

[10] Windows Multipoint is not a remote desktop technology but rather a framework supporting multiple mouse cursors like MPX does, so it is included here as well.
[11] There is a software development kit available, but applications available from Microsoft are proprietary.
[12] Mikogo allows changing the presenter role on the fly, but only one user at a time can share her screen.
[13] TeamViewer does not support annotations on all operating systems.
[14] In Adobe Connect, only the presenter can draw annotations.
[15] In Adobe Connect, client window sharing does not work on Unix.
[16] SharedView provides some basic floor control mechanism, but no concurrent user interaction.
[17] PCAnywhere supports multicast, but users cannot control the remote desktop in this mode.
[18] TeleTeachingTool offers annotations only for the lecturer, not for connected clients.

	View	Control	Multi-User Operation	Annotations	Client Window Sharing	Multicast	Autom. Server Discovery	Cross-Platform Client	Source Code Available
THINC	✓	✓	(✓)[19]	-	-	-	-	✓	✓
BASS	✓	✓	(✓)[20]	-	-	✓	-	✓	-
MAST	✓	-	-	-	✓	✓	✓	-	✓

3.3 Conclusion – Motivation for CollabKit

After the examination of usable related work was completed, it was found that almost all of the considered software products support basic remote view and control features, but it also was evident that multi-user or multicast support is relatively scarce. In particular, it became clear that *none* of the considered software products could fulfil *all* of the requirements posed. For instance, three VNC based solutions support multicast, but they all lack basic remote control support (see table 2). Similarly, some of the products considered in subsection 3.2.4 to some extent support multicast data transmission, but none of them feature real *concurrent* multi-user support. In fact, MPX mentioned in subsection 3.2.1 is the only solution supporting multiple users interacting concurrently with mice *and* keyboards. Windows Multipoint has support for multiple cursors, but lacks support for multiple keyboard foci. All other products just support turn-taking of users.

Since it was decided within the requirements analysis that complete[21] multi-user support is needed and that it should provide not only turn-taking, but concurrent collaboration of participants, the use of MPX for further work was considered somewhat mandatory. However, the other considered X11 based solutions were found to poorly satisfy other functionality and performance requirements: None of them provide annotations, client window sharing, multicast or server discovery[22]. x2x and xtv lack even basic features, NX is only partly open source. Furthermore, within X11 an application can always just be connected to a single X server, without special measures it is thus impossible to display and remote control an application from two different remote computers. XMX provides such features to share applications among several clients, but its codebase is severely

[19] There is an extension of THINC [83] that offers basic floor control, but no concurrent user interaction.
[20] BASS provides some basic floor control mechanism, but no concurrent user interaction.
[21] Complete multi-user support means multiple cursors *and* keyboard foci.
[22] There is no built-in server discovery, but a stand-alone X display manager tool using XDMCP [110].

outdated and would probably require considerable work to be run on today's X Window System implementations. Another general issue with X11, especially on intermittent connections, is that its network protocol is a *stateful* one: If the connections fails, the application loses its X server and terminates. Finally, the X11 network protocol involves many round trips, which hampers performance on high-latency links [111].

Taking these findings about X11 into account, it was concluded that combining MPX with another remote desktop technology would be a more promising approach. The other possible solution would have been to start with an existing remote desktop technology and implement concurrent user interaction support there, but taking into account the time MPX development needed [34], it was decided to build upon the work done and implement missing features in the remote desktop software instead. It was also clear that for interfacing with MPX, this remote desktop software would have to be modified. Thus, only products with available source code were furthermore eligible. An in-depth comparison of software that came into consideration is shown in table 5. Particularly multicast support is examined in more detail.

Table 5: In-Depth Comparison of Remote Desktop Software interfaceable with MPX.

	View	Control	Multi-User Operation	Annotations	Client Window Sharing	Multicast	Multicast Error Handling	Multicast Flow Control	Unicast and Multicast in parallel	Autom. Server Discovery	Cross-Platform Client
Multicast-VNC	✓	-	-	-	-	✓	-	-	-	-	✓
TightVNC	✓	✓	-	-	-	-	-	-	-	-	✓
xf4vnc	✓	✓	-	-	-	-	-	-	-	-	✓
MetaVNC	✓	✓	-	-	-	-	-	-	-	-	✓
Tele-Teaching-Tool	✓	-	-	(✓)[23]	-	✓	-	(✓)[24]	✓	-	✓
Vino	✓	✓	-	-	-	-	-	-	-	✓	✓

[23] TeleTeachingTool offers annotations only for the lecturer, not for connected clients.

[24] The new implementation [50] of TeleTeachingTool uses RTP instead of VNC and thus supports multicast flow control.

	View	Control	Multi-User Operation	Annotations	Client Window Sharing	Multicast	Multicast Error Handling	Multicast Flow Control	Unicast and Multicast in parallel	Autom. Server Discovery	Cross-Platform Client
x11vnc	✓	✓	-	-	-	-	-	-	-	✓	✓
Collaborative VNC	✓	✓	(✓)[25]	-	-	-	-	-	-	-	✓
SharedApp-VNC	✓	✓	-	-	✓	-	-	-	-	-	✓
xrdp / rdesktop	✓	✓	-	-	-	-	-	-	-	-	✓
VNCast	✓	-	-	-	-	✓	-	✓	-	-	✓
THINC	✓	✓	(✓)[26]	-	-	-	-	-	-	-	✓
MAST	✓	-	-	-	✓	✓	-	-	(✓)[27]	✓	-
CollabKit	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

As stated above, the requirement for remote desktop software to appear in table 5 was that source code be available so that modifications could be made. Another prerequisite was that there should exist a server implementation for the X Window System that could be interfaced with MPX.

The fundamental question at this point was which remote desktop software would be suited best as a starting point to implement missing features. As can be seen in table 5, the majority of the remaining entries is based on VNC, only one on RDP and three on other protocols.

Looking at the entries that support multicast (MulticastVNC, TeleTeachingTool, VNCast and MAST), it is noticeable that none of them supports remote control, they are all view-only. While the missing remote control features could possibly be added with maintainable effort, the lack of proper multicast flow control and error handling is a more serious problem. This is important because IP multicast is based on UDP instead of TCP and thus provides no built-in flow control nor reliable data transmission. Out

[25] Collaborative VNC supports basic floor control, but no real concurrent multi user control.
[26] There is an extension of THINC [83] that offers basic floor control, but no concurrent user interaction.
[27] MAST supports unicast via additional software.

of the considered multicasting solutions, only the »new« TeleTeachingTool and VNCast potentially provide multicast flow control since they are based on RTP, but they still lack multicast error handling. Furthermore, only two multicast systems provide an additional unicast communication channel, but in either case this is *not* a reliable TCP connection but merely a unicast UDP connection for clients that do not support multicast. However, a reliable connection can be useful for data sensitive to packet loss, for instance remote control input data sent to the server or sparse but important server-to-client messages where it makes no sense transmitting them via multicast.

Because of these shortcomings of existing multicast remote desktop software, it was decided to implement multicast support from scratch, extending an existing unicast remote desktop software. This way proper multicast flow control and error handling could be implemented in a clean fashion while keeping the existing unicast communication paths for loss-sensitive data and as a fallback.

At this point, nine entries in table 5 were still eligible for further work: seven solutions based on VNC, one product using RDP (xrdp/rdesktop) and one using a custom protocol (THINC). Since it was found to be important that the resulting system utilize a widely used protocol in order to allow legacy non-multicast clients to connect as well, THINC was then ruled out precisely because it relies on a custom protocol that is not in widespread use. That left VNC or RDP as the basic protocols to be extended with multicast support.

The fundamental question now was whether to extend one of these protocols or to *add on* multicast support atop using another protocol. For the latter case, the Realtime Transport Protocol RTP [117] was examined, but found to be overly complex for the intended use case. RTP has a lot of features that are needed for proper transmission of audio data, but are of little use for multicasting of simple image data: these are anti-jitter buffering, reordering of packets, timestamps and statistics gathering. Buffering of incoming data in order to compensate possible jitter is essential for audio data, but not really necessary for image data. Since both VNC and RDP image payloads are split up into relatively small packets that are tagged with size and position information, the order in which packets arrive is also irrelevant. Again, timestamps are essential for transmission of audio data, but of little use for image data. Finally, the extended statistics gathered by the associated RTCP protocol like jitter and round-trip delay time also are useful mainly for audio transmissions. The only value of fundamental interest for transmission of image data is the packet loss ratio, which can be obtained by using simple sequence numbers. It was therefore concluded that *extending* one of RDP or VNC with simple but adequate multicast support would be a better suited approach resulting in a less complex system in the end.

VNC and RDP use – as opposed to X11 – *stateless* network protocols: all state is stored at the server side. When a connection terminates unexpectedly, the application keeps on running on the server. The client can simply reconnect and continue working where it has stopped. Another benefit of storing all state at the server is that the client application can be kept very simple: it just has to display image data and forward input

data to the server. This way applications written for one platform can easily be remote controlled from a different one, provided a VNC or RDP client implementation exists. Both systems basically send server screen updates as – optionally compressed – image raster data. Aside from these similarities, VNC and RDP differ in one fundamental aspect, which is the way server-to-client updates are delivered: RDP uses a server-push approach whereas VNC employs a client-pull update mechanism[28] [107, 115]. The difference between the two approaches mainly shows up on high-latency connections: because of the additional round trips involved, simple client-pull systems perform worse than server-push systems in such environments [123]. However, as shown in section 4.2.3, this issue can be worked around easily. It was concluded that both systems would provide a usable base to build upon, but in the end VNC was chosen over RDP because of its greater simplicity [115, 107] and wider deployment [42, 41].

With VNC decided upon as the system to extend, this left seven specific software products as a possible starting point for future implementation, namely TightVNC, xf4vnc, MetaVNC, Vino, x11vnc, CollaborativeVNC and SharedAppVNC. The latter one was then excluded for the same reason as THINC was: it uses a nonstandard RFB protocol that is incompatible with standard VNC.

Out of the remaining six candidates, x11vnc was identified as the most promising starting point. Together with Vino it is the only server software that is able to share an *existing* user session. All other products either only support creating a new VNC session different from the local user's one[29] or can only export an existing session after cumbersome installation requiring superuser privileges[30]. On the one hand, this ability to easily share an existing user session was required by the use cases, on the other hand it was found to be the only way to share accelerated OpenGL applications: TightVNC, xf4vnc, MetaVNC and CollaborativeVNC all use an internal X server to create a new session. However, these internal X server implementations all lack the extensions needed for hardware-accelerated OpenGL. Out of the two products that allow sharing existing sessions, x11vnc was then chosen over Vino because it is not tied to a certain desktop environment like Vino. In addition, x11vnc already provides service announcement, via Zeroconf [82], the ability to allow or disallow full or view-only access to the shared desktop and encrypted communication.

Regarding the client application that participants would use to connect to the central desktop, it was found that none could meet all the requirements posed: The Multicast-VNC and TeleTeachingTool clients are both written in Java and hence cross-platform, but they do not allow remote control and can only receive their custom multicast VNC protocols, which were ruled out because they lack proper flow control and error handling. SharedAppVNC is the only VNC application that supports client-to-server window sharing, but it uses a nonstandard modified RFB protocol and needs a custom server appli-

[28] Server-push means that the server pushes out updates as soon as they are available, client-pull denotes a mechanism where the client asks the server for updates.

[29] TightVNC and its offsprings MetaVNC and CollaborativeVNC.

[30] xf4vnc provides an X server plugin, but its installation needs superuser privileges and non-trivial configuration.

cation. Out of the mass of standard VNC client applications, only UltraVNC, RealVNC and TightVNC provide cross-platform Java applets and only the latter two ones have native client implementations for Unix and Windows available. However, all of them lack automatic server discovery, client-to-server window sharing and multicast support. It was concluded that extending existing client implementations would not be easily feasible because on the one hand the native implementations use different code bases for each platform, on the other hand it was felt that implementing the needed features, especially client-to-server window sharing, would be overly hard using Java. Instead, it was decided that writing a client application with the needed features from scratch would be a more promising approach, especially when using the LibVNCClient library [21] which abstracts a lot of VNC protocol internals using a simple API.

4 Design of a Multi-User Multicast Collaboration System

After the examination of related works was finished and a promising candidate for further work identified, it was necessary to find out *what* had to be implemented exactly and *how* it should be designed. This section deals with this process and documents how design and implementation goals were deduced from requirements and already present features. Questions that arose during planning and the resulting final design of the written software are presented here as well.

4.1 CollabKit Needed Functionality

Out of the requirements identified in section 2.2 the chosen *server* software to build upon, x11vnc, already fulfilled some while also lacking fundamental needed features. With regard to the *client* application it became clear that it had to be written from scratch.

The different non-functional and functional requirements a real-time collaboration system should meet were identified in subsection 2.2. This also included tracing back the higher-level non-functional requirements to low-level functional requirements where applicable. The resulting set of »pure« functional requirements and functional requirements pertaining to non-functional ones was summarised in figure 6 on page 13.

Functional requirements regarding specific features are:

- The ability to view the shared desktop.
- Functionality to remote control the shared desktop.
- Support of multiple mouse cursors and keyboard foci on the shared desktop.
- Functionality for participants to export their own windows to the shared desktop.
- A graphical annotation mode on the shared desktop.

The following functional requirements were deduced from non-functional ones:

- Multicast transmission of image data. This greatly helps to improve *scalability*.
- A basic access control. One aspect of *security*.
- Encryption of occurring communication. The other aspect of *security*.
- Automatic server discovery. Besides an intuitive GUI this also is important for *ease of use*.
- The client application should be *cross-platform*. A high level of *platform independence* allows for a greater variety of client systems and thus for a broader user base.

Of these fundamental functional requirements the server application x11vnc does already fulfil some, mostly basic ones. These are: functionality to *view* and *remote control* a

central desktop, the ability to change between full or view-only *access* to this desktop, service announcement via Zeroconf, simple access control and encryption. What is missing is a graphical *annotation* mode, fully *concurrent multi-user operation* and finally the support for multicasting of image data.

As described in section 3, no existing VNC client application could be found that would suit the needs posed by the requirements analysis. Thus – and because performance measurement features were needed for evaluation – it was decided to write a custom client application from scratch, incorporating the needed features.

Taking these considerations into account, the following implementation goals could be deduced:

- **Regarding Multi-User Support:**
  - **Multi-Pointer extension of x11vnc by interfacing it with MPX in order to provide concurrent multi-user operation.**
  - **Multi-Pointer graphical annotation functionality on the shared desktop.**
  - **A Cross-platform client application that uses a simple graphical user interface, supports automatic server discovery, and provides performance measurement functionality.**
  - **Functionality to export a participant's window to the shared desktop at both the client and server side.**
- **Regarding Multicast Transmission of Image Data:**
  - **A multicast session setup mechanism to avoid the use of hard-coded defaults.**
  - **A mechanism to deal with the different VNC pixel-formats and encodings that clients can request.**
  - **A resource-saving update scheme for serving data requested by a large number of clients.**
  - **Mechanisms to deal with the fact that multicasting requires the use of UDP datagrams as opposed to TCP byte streams.**
  - **A simple yet scalable multicast flow control scheme.**

The individual implementation goals and the design concepts developed to achieve them are discussed in greater detail below.

4.2 Multi-User Support

MPX Changes Regarding Device Handling

Because the first two implementation goals, »multi-pointer extension of x11vnc« and »multi-pointer graphical annotations« both depend on the functionality provided by

MPX, it is important to first clarify how MPX is designed and how it differs from the traditional X Window System.

Traditionally, there has always just been one mouse pointer and one keyboard in the X Window System. Even when several physical devices were plugged in, the were all mapped to the so-called *core pointer* or the *core keyboard*, respectively. These *logical* devices send so-called *core events* when their status changes, for instance when a mouse is moved or a key got pressed. In addition to this traditional common input system, newer X Window System incarnations also implement the so called *XInput* extension which adds more features with regard to device handling, like providing pressure or tilt information generated by devices like graphics tablets. This information is encapsulated in special *XInput events* different from common core events. As opposed to core events, which are supported by all X11 applications, reception of XInput events is only supported by few applications, mostly graphics software. Still, XInput devices can be mapped to core pointer or core keyboard in order to make them generate core events. However, additional information like tilt or pressure is lost this way.

MPX changes this traditional input handling paradigm in a fundamental way. Instead of a single core pointer and core keyboard, it introduces the concept of several *logical master devices*. Each master pointer is represented by a cursor on the screen, each master keyboard by an input focus. Physical input devices are represented by so-called *slave devices*, which can be attached to and removed from master devices at will. An example of the resulting hierarchy of input devices is depicted in figure 10.

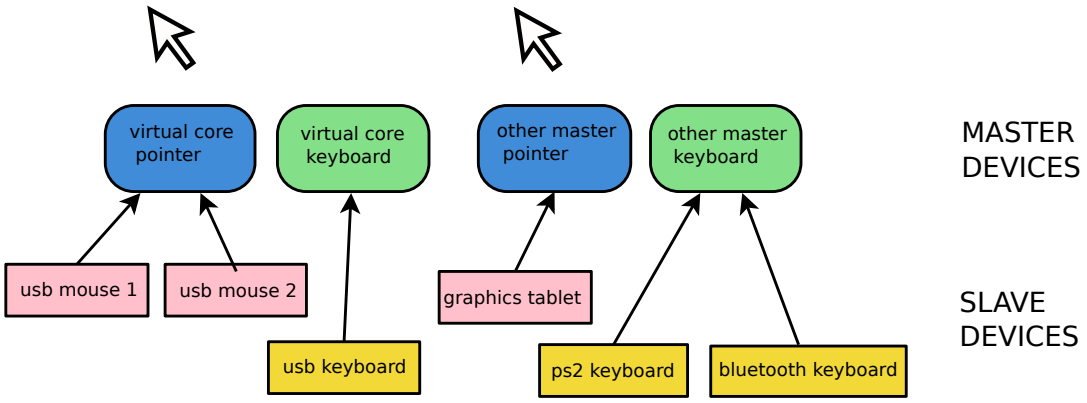


Figure 10: Example input device hierarchy with MPX.

In this hierarchy, the master device layer is what is important to applications in terms of input handling: on the one hand, master devices send traditional core events for legacy applications to receive. On the other hand, they also send XInput events that contain more information, especially the indication *which* device generated the event. Thus, MPX basically is a substantial extension of the original XInput device handling infrastructure.

Making existing applications aware of multiple pointers therefore essentially means changing their input handling from the traditional core event infrastructure to the newer XInput system.

4.2.1 Concurrent Multi-User Operation

Concurrent multi-user operation of the shared desktop was found to be beneficial for all use cases considered in section 2 because it enables participants to *jointly* interact with objects on the shared desktop *at the same time*. This could be achieved by extending the server application x11vnc and interfacing it with MPX: when a client connects, it gets its own MPX master pointer and keyboard focus which can be operated independently from other MPX master devices. It was found that using differently coloured cursors for mouse pointers is imperative in order not to confuse users.

Therefore, three basic features that had to be implemented were identified:

- When a new client connects, a new mouse-keyboard pair of MPX master devices has to be created. On connection termination, both master devices have to be removed. It was found that creation and removal of master devices can be achieved relatively easy by using library calls provided by libXi [23], the XInput support library within the X.Org X server distribution.
- To properly route input events to the client's assigned master devices, all functions, variables and data structures in the x11vnc sources that deal with client input had to be extended to be device-aware. This is documented in more detail in section 5. For injection of client input into the assigned master devices, the XTEST extension [85] can be used, that is, the device specific calls provided by the libXtst [24] library.
- Coloured and labelled cursor images can be created *on the fly* using the vector graphics library Cairo [6]. This way, user names can be displayed beneath the cursor, a feature not available when using pre-rendered cursor images. The created cursor images can then be assigned to some master pointer using the X11 cursor management library libXcursor [22].

4.2.2 Multi-User Graphical Annotations

For making graphical annotations on the shared desktop possible with multiple pointers at once, it was decided to extend the annotation tool Gromit with multi-pointer support.

Gromit [15] stands for »*GR*aphics *O*ver *MI*scellaneous *T*hings«. With Gromit, graphical annotations in different colours can be drawn onto an X display. It normally runs in the background of the current session, but can be activated by hitting a certain hot-key, enabling the user to draw onto the screen as shown in figure 11. Gromit supports special features like tilt or pressure recognition when using special input devices like graphics tablets.

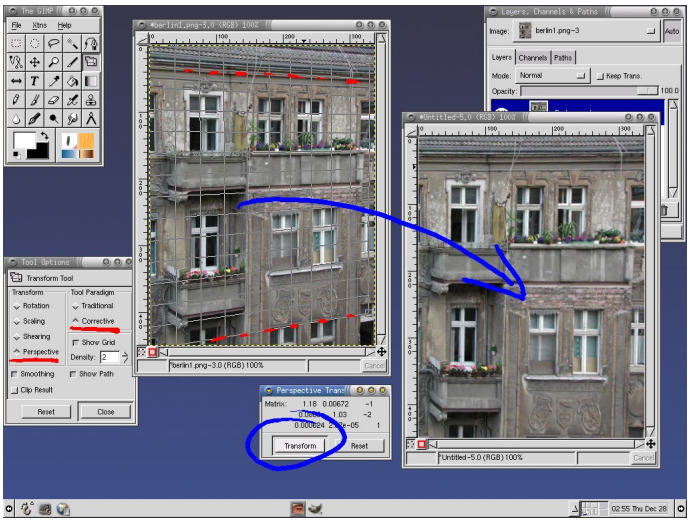


Figure 11: Graphical annotations using Gromit [16].

Since the widget toolkit used by Gromit, GTK+ [17], has MPX support in its most recent development versions, the remaining task was to change the application itself to be aware of multiple pointers. The goal was to enable users to draw annotations concurrently while other users keep on working normally. Like all legacy applications, Gromit only has the notion of a single pointer and keyboard. The internal data structures and functions in the program are laid out accordingly. In order to enable drawing with multiple pointers in Gromit, these internal data structures and associated code had to be modified so that relevant data is stored *per device*.

The steps taken in order to extend Gromit with support for multiple input devices were:

1. Identification of variables and data structures that belong to the context of an input device.
2. Refactoring of these variables and data structures into a generic device-specific data structure, which will get instantiated dynamically *per device*.
3. Adaptation of associated code to be device aware and to handle the new per-device data structures.
4. Adaptation of code that does not deal directly with the identified data structures or variables, but nonetheless deals with input devices. For instance, this is code that directly changes the mouse cursor or keyboard state in the X server. Instead of just changing the state of core pointer or core keyboard, the code had to be modified to change *several* master devices either *at once* or *individually*. This distinction was supposed to bring additional complexity into the code.

The above steps were taken in order to extend Gromit specifically, but surely are applicable to most other programs. What was changed exactly is documented in greater

detail in section 5.

4.2.3 Cross-Platform Client Application

The specific implementation goal for the client application was the integration of all client functionality in a simple graphical user interface. This meant that common VNC client functionality, server discovery and client-to-server window sharing controls along with performance measurement means were to be implemented in a single application.

Regarding the needed VNC client functionality it was found that using the LibVNCClient library that comes with LibVNCServer [21] would be the best approach. This library hides most of the complexities of VNC session setup and tear-down behind a simple API. It also handles communication during an active session and provides the application using it with an in-memory framebuffer representation of the remote screen.

As mentioned in subsection 3.3, the Remote Framebuffer Protocol used by VNC does not perform very well on high-latency links because of it's client-pull update mechanism: after requesting a framebuffer update, a conventional VNC client waits for the server's answer before issuing a new request. This results in a very low rate of updates from the server on high-latency links. Fortunately, this issue can be worked around quite simply: by letting clients continuously issue framebuffer update requests, a rate of server-to-client updates similar as in a server-push environment can be achieved. Figure 12 illustrates this: whereas with conventional VNC just two updates arrive at the client side, a client continuously requesting framebuffer updates receives six updates in the same time.

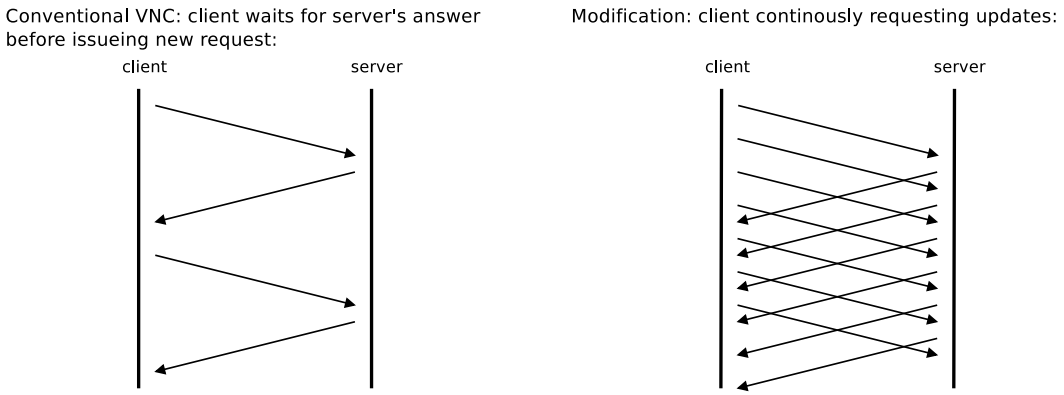


Figure 12: Protocol sequence diagrams comparing conventional VNC with a modification that lets clients continuously request updates.

To display the in-memory image provided by LibVNCClient, to gather input to be sent to the remote side and to provide the user with client-to-server window sharing controls, a cross-platform graphical user interface was needed. Since LibVNCClient is written in C, it was decided to use a C or C++ widget toolkit to implement the GUI. Out of

the multitude of available toolkits, the *cross-platform* C++ solution wxWidgets [67] was then chosen. The main reason for this choice was that unlike other toolkits, wxWidgets provides *native* look and feel on all supported platforms because it uses the platform's native API instead of emulating the GUI.

In order to enable the user to connect to a server quickly and without unneeded configuration, automatic server discovery had to be implemented as well. Since the server application x11vnc already does service announcement using Zeroconf [82], it was logical to build upon that and use Zeroconf service discovery on the client side as well. For that purpose, the wxWidgets service discovery class wxServDisc [66] was found to be suited best because like wxWidgets itself, it too has cross-platform support.

Finally, some performance measurement functionality was needed in the client application to be able to properly compare multicast against unicast data transmission. It was found that measuring achieved *throughput*, *latency* of updates and multicast *loss ratio* at the client side would be adequate.

4.2.4 Client-to-Server Window Sharing

Because VNC is used for distributing the server's screen to connected clients, it was obvious to use the same technology to export client windows to the central desktop. This meant that regarding this functionality, the actual server machine would need to act as a VNC *client* receiving windows whereas the participant's computers would act as VNC *servers* exporting windows.

On the server machine, this can be achieved through running a VNC client in *listening* mode. As the name implies, in this mode a VNC client listens for an incoming connection originating from a VNC server instead of initiating the connection itself. Therefore, this mode is often also called »reverse VNC«. In order to be able to receive several windows from different participants, the listening VNC viewer application must somehow be able to handle several connections at once, either by forking itself or using multi-threading. Resizing the viewer window in case the exported participant's window is changed in size was found to be possible by making use of the DesktopSize pseudo-encoding [115, p. 43].

On a participant's computer, client-to-server window sharing is possible by simply running a VNC server software that supports sharing single windows instead of the whole screen and that is able to connect to a listening VNC viewer. For Unix-based systems, using x11vnc was found to be the best approach. Together with SharedAppVNC, it supports the aforementioned two features, but unlike SharedAppVNC, x11vnc uses unmodified standard VNC and does not need superuser privileges for installation. For computers running Microsoft Windows, RealVNC, UltraVNC and TightVNC were eligible. TightVNC was then chosen because it has the advantage of supporting the throughput-efficient »tight« VNC encoding. Finally, as mentioned in subsection 4.2.3, the client-to-server window sharing functionality needs to be easily accessible from within the client application's user interface.

4.3 Multicast Transmission of Image Data

As stated in section 3.3, it was decided to extend the remote desktop technology VNC with support for multicasting of image data. This way the common unicast communication paths can be used for loss-sensitive data while bulky image data is transmitted to clients via multicast, providing significant channel capacity savings when several clients are connected.

This section first outlines the basic differences between unicast, broadcast and multicast data transmission and then explains which problems this posed in the design process of a multicast VNC extension. After documenting how these problems were solved, the resulting final design of the *MulticastVNC* extension is presented.

Basic Principles

In the context of *Internet Protocol* networking, a unicast IP address identifies a *single* IP interface whereas a broadcast address identifies *all* IP interfaces on the local subnet. Multicasting, on the other hand, means addressing a certain *set* of IP interfaces. This set can contain any number of recipients. Unlike broadcasted datagrams, a multicast datagram is only received by the IP interfaces belonging to this so-called *multicast group*. Additionally, broadcasting is normally limited to the local subnet, whereas multicasting can be used on LANs as well as across WANs.

Both broadcasting and multicasting have in common that each datagram is sent to all receivers *once* instead of being delivered to each one individually. This constitutes an advantage over unicast addressing when the same data is to be delivered to multiple recipients. Since distributing the screen contents of a single server to multiple clients is exactly such a scenario, it is obvious to use multicasting instead of unicasting in this case. However, the *Remote Framebuffer* protocol [115] used by VNC does only support unicast data transmission, relying on the *Transmission Control Protocol* TCP at the transport layer. It was therefore decided to extend this protocol to support multicast transmission of screen contents. Since other messages defined by the protocol do not consume nearly as much throughput capacity as these server-to-client framebuffer updates do, it was deemed adequate to let the extension only transmit framebuffer update messages using multicast. Multicasting of other RFB messages was found to not provide a noticeable benefit because of their small size.

As suggested by the RFB protocol specification [115, p. 5], the multicast VNC extension was realized by introducing a new pseudo-encoding. This way the protocol can be extended in a backward compatible fashion: a client tells the server that it supports a certain extension by requesting the associated pseudo-encoding. If the server also supports the extension, it will answer with an extension-specific confirmation, otherwise it will simply ignore the request.

Before the final design of the multicast VNC protocol extension is presented, the next section documents what questions arose during the design process and how they were

solved. Meaning and structure of mentioned RFB messages are explained in detail in the Remote Framebuffer Protocol specification [115].

4.3.1 Delivery of Multicast Group Address to Clients

Within the Internet Protocol, multicast addressing is realized by reserving a certain subset of the IP address space for multicast destinations [77, 93]. Each of these multicast IP addresses identifies a *multicast group*. In order to receive messages destined for this multicast address, an application has to first *join* the multicast group, otherwise those messages will be discarded. Therefore, the receiving application has to know about the multicast destination the sender uses, either by convention or by other means.

In order to be as flexible as possible, it was decided to not stick to a fixed multicast address and port, but instead use the aforementioned server confirmation of an extension requested via a pseudo-encoding to convey the needed information to a client. Not relying on a fixed address makes it possible to operate several multicasting VNC servers in the same network and also permits the use of advanced techniques like Multicast Administrative Scoping [106].

Thus, a client supporting the multicast VNC extension will ask the server if it also does by requesting a newly defined *MulticastVNC* pseudo-encoding within its *SetEncodings* message. If the server supports the extension, it will answer with a *FramebufferUpdate* server-to-client message consisting of a single rectangle with its encoding set to *MulticastVNC*. This rectangle then contains the multicast destination IP address and UDP port chosen by the server. If IPv6 multicasting is to be used, the client can request a newly defined *IPv6MulticastVNC* pseudo-encoding, respectively. The general session setup scheme is the same.

4.3.2 Different VNC Pixel-Formats and Encodings

Within traditional unicast VNC, every client is allowed to choose its preferred pixel-format and encoding. The VNC server then *must* provide the requested pixel-format and *can* choose to supply the client with pixel data in the specified encoding. This works perfectly well if each client has its own communication channel as is the case with traditional VNC, but poses a problem if several clients share a single communication channel, as is the case when using multicasted VNC. Three possible solutions to this problem were taken into account:

One possibility was to use a *single fixed* pixel-format and encoding pair. This either means simply using a predetermined format-encoding pair *by convention* or letting the server choose one which clients would have to support in turn. The former approach was deemed too inflexible as it pins down the whole multicast extension to a single pixel-format and encoding. The latter approach would require some form of handshake between client and server to negotiate what should be used. While this would allow the server to choose a certain pixel-format and encoding, it is overly complicated and still

forces clients to use this chosen format-encoding pair. Both approaches do not comply with the thin-client philosophy of VNC that tries to relieve the client of as much burden as possible: they both force clients to use a certain pixel-format and encoding. Therefore, the possibility of using a fixed format-encoding pair was dismissed.

Another possible solution was to use a fixed pixel-format, but allow different encodings. This is feasible because each rectangle carrying pixel data also carries information about the encoding used. While this approach is a little more flexible than the first possible solution, it still forces clients to use a certain pixel-format and was therefore dismissed for the same reasons as mentioned above.

The third possible solution taken into account was to allow several pixel-formats and encodings. This does not force clients to use a certain pixel-format and thus adheres to the thin-client paradigm of VNC. On the other hand, it makes it necessary to distinguish rectangles with different pixel-formats. It was found this could be done by either starting a new multicast session (with new destination addresses and/or ports) for each pixel-format to be served or by tagging framebuffer updates with information about the pixel-format they use. However, starting new multicast sessions becomes a serious problem when several multicast VNC servers are active on the same subnet: when servers are free to choose their multicast destination address, it is possible that two or more servers multicast to the same destination address this way. Routing also becomes more difficult. The latter approach of tagging framebuffer updates was therefore deemed superior although it requires to tag either rectangles or whole framebuffer updates with an additional pixel-format field, introducing some overhead.

In the end, this third solution to the problem was chosen, simply because it adheres best to VNC's thin-client philosophy of trying to put as much workload as possible off the client onto the server.

4.3.3 Accumulation of Update Requests

The next question was whether to send multicast framebuffer updates whenever the screen contents change (server-push) or each time clients request them (client-pull).

The solution decided upon was to do a mixture of both: the server sends out a multicast framebuffer update to all registered multicast clients after expiration of a certain non-zero time span, its multicast update interval. However, it does only send the update if at least one multicast client requested an update before. This way, the server cannot be overloaded with too many requests it would have to serve. Additionally, no updates are sent when no client is interested in them.

4.3.4 Datagrams Instead of Byte Streams

Multicasting Uses Datagrams with a Maximum Size

Since the *Transmission Control Protocol* TCP is a connection-oriented transport protocol using the concept of a one-to-one connection between two hosts, it cannot be used for multicasting. Instead, the connectionless *User Datagram Protocol* UDP is normally used at the transport layer when doing multicasting. Unlike TCP, UDP does not provide a reliable byte stream. Instead, it simply features sending and receiving of single messages referred to as *datagrams*. The way the UDP message header is designed limits the maximum payload of a UDP datagram to roughly 65 kilobytes.

In consequence this means that a VNC framebuffer update sent via multicast very probably has to be split up into several smaller updates that each fit into a UDP datagram. To properly associate those partial updates with a whole logical framebuffer update, it is useful to number both partial and whole update.

Multicasting Uses Datagrams Which can Arrive Out of Order, be Duplicated or Lost

In addition to having a fixed maximum size, UDP datagrams are also not guaranteed to arrive at their destination in the correct order, to arrive only once or to arrive at all. For the intended use case, the transmission of pixel data, lost messages are the biggest problem. Duplicate messages or out-of-order messages are less problematic due to the way the RFB protocol indexes pixel data by putting it into rectangles that carry position and size information.

However, in order to enable clients to detect lost messages, adding sequence numbers to partial and whole logical framebuffer updates is mandatory. An additional requirement regarding error *discovery* is that clients must be able to distinguish between two cases: either no messages are received because there were no updates to send or nothing is received because all messages got lost. Only the latter case is important with regard to error ratio estimation, the former is not. The solution to this problem is to let the server send heartbeat messages at its multicast update interval when no actual updates are pending but at least one client requested an update. This way, clients are still able to rate the connection's quality to some extent even when nothing changed in the server's framebuffer and no updates were pending. They have then, however, to get to know the server's multicast update interval somehow. This can be done on session setup when the multicast destination address is sent.

Concerning error *handling*, these possible approaches were taken into account:

The first considered approach was to just let clients calculate an average multicast loss ratio. This way, loss of single datagrams is ignored until the loss ratio reaches a certain boundary value. The downside to this simplistic approach is that clients have no knowledge about which lost message's sequence number maps to which region of the

server's framebuffer. Therefore, they instead have to resort to the request of a full non-incremental multicast framebuffer update.

Another idea was to simply let clients acknowledge each datagram and let the server retransmit those that lack an acknowledgment by every registered multicast client. While this would certainly be feasible, it is not an ideal solution simply because of the sheer amount of *ACK* messages that would increase with every client joining the multicast VNC session. Although there are approaches to alleviate this problem of an *ACK implosion* using hierarchical acknowledgement schemes [103, 114] or ACKs distributed over time [92], there are some hints [112, 121] that using *negative acknowledgments* or *NACKs* is a better suited approach for multicast data transmission: Clients tell the server which datagrams got lost instead of acknowledging each and every received datagram. As mentioned above, clients are able to detect lost messages by noticing missing sequence numbers. They can then request retransmission of missing packets by sending NACK messages to the server. This way there is a lot less traffic than with the ACK solution. In fact, there only is additional traffic if clients notice lost datagrams. It was decided that this would be the most adequate approach to deal with the problem of lost messages. Other works concerned with making multicast more reliable also rely on some form of NACK mechanism [75, 76, 87].

4.3.5 Multicast Flow Control

Flow control is the process of managing the rate of data transmission between network nodes to prevent a fast sender from overwhelming a slow receiver. Since UDP does not provide built in flow control like TCP does, an application layer multicast flow control scheme had to be integrated into the *MulticastVNC* protocol extension.

Flow control mechanisms can be designed as simple open-loop or more sophisticated closed-loop control systems. In an open-loop control system, there is no feedback between sender and receiver, send rate calculation is solely based on the sender's assumptions about the network's state. Obviously, such a control scheme is unable to adapt to changing environments. Therefore, a closed-loop flow control system that uses feedback from receivers to adapt the sender's data transmission rate was deemed the only viable approach. Such feedback-based flow control schemes can be *credit-based* or *rate-based*:

In the credit-based or *sliding-window* approach, the receiver specifies a limit on the number of packets that the sender may send without further permission from the receiver. Typically, this permission is not sent as an explicit message[31] but given by acknowledging received messages so that the sender can advance its sliding window. Thus, this form of flow control uses *ACK* messages as means of receiver-to-sender feedback. Exemplary multicast transport protocols using this form of flow control are [114], [92] and [103].

[31] An example of credit-based flow control where explicit send permissions are given is a simple stop-and-go scheme: A slow receiver can halt the sender by sending a stop message and resume transmission with a start message. In this special case of credit-based flow control the send credit is either zero or infinity.

In contrast, rate-based flow control mechanisms do not maintain some form send credit but directly adapt the rate at which the sender is transmitting data. Again, this can be handled explicitly by sending special control messages (like RTP [117] does) or implicitly by making use of information conveyed with error handling messages. Typically, rate-based flow control schemes are used with *NACK*-based error handling mechanisms: a message retransmission request by a receiver is interpreted as an indication to lower the send rate. This form of flow control is employed by related works such as [101], [87] and [122].

Since a NACK-based approach was chosen for *MulticastVNC* to handle data transmission errors, a rate-based flow control scheme was considered most suitable. *MulticastVNC* uses a modified form of the send rate adaptation algorithm proposed in [122] and [121].

The original algorithm described in [122] adapts the sender's rate of data transmission based on a timer and the receipt of *significant* NACK messages: The sendrate R is additively *increased* by a certain increment value I on expiration of a timer T as described by equation 5:

$$R_i = R_{i-1} + I \tag{5}$$

The value of T is dependent on R and gets calculated on every change of R:

$$T = \frac{c}{R} \tag{6}$$

where c is a constant value that should be bigger than the sum of the sender's and receiver's buffer capacity.

Also, on every m consecutive rate increases *without* intermittent rate decreases, the increment I itself is multiplicatively increased by a constant factor n:

$$I_i = I_{i-1} * n \tag{7}$$

The send rate R is *decreased* only on receipt of *significant* NACK messages. The flow control scheme described in [122] distinguishes between NACKs that indicate that the current send rate should be lowered (significant NACKs) and NACKs that convey duplicate information concerning flow control and thus are meaningless:

- Of all NACKs for messages sent at same transmission rate arriving at the sender, only one is significant. The others are meaningless because they all indicate that the transmission rate at which the corresponding messages were sent was too high for at least on receiver.
- A NACK for a message sent at a higher transmission rate than the current one is also meaningless, it does not indicate that the *current* transmission rate is too high.

To distinguish between significant and meaningless NACKs, the approach proposed by [122] logs the transmission rates at which messages were sent and additionally uses a flag per transmission rate indicating whether a transmission rate has already been decreased or not:

On receipt of a NACK, the sender looks up the transmission rate of the corresponding message in aforementioned log and checks its flag. If the flag indicates that this transmission rate has not been decreased *and* the current transmission rate is higher than or equal to the logged transmission rate, the NACK is considered significant. Thus, the current transmission rate is decreased and the flag is set.

The exact rate decreasing on receipt of a significant NACK is described by equation 8:

$$R_i = R_{i-1} * \frac{1}{n} \tag{8}$$

The parameter n denotes a constant rate decreasing factor and is the same as in equation 7.

Finally, the increment value I is also decreased on receipt of a significant NACK:

$$I_i = I_{i-1} * \frac{1}{n} \tag{9}$$

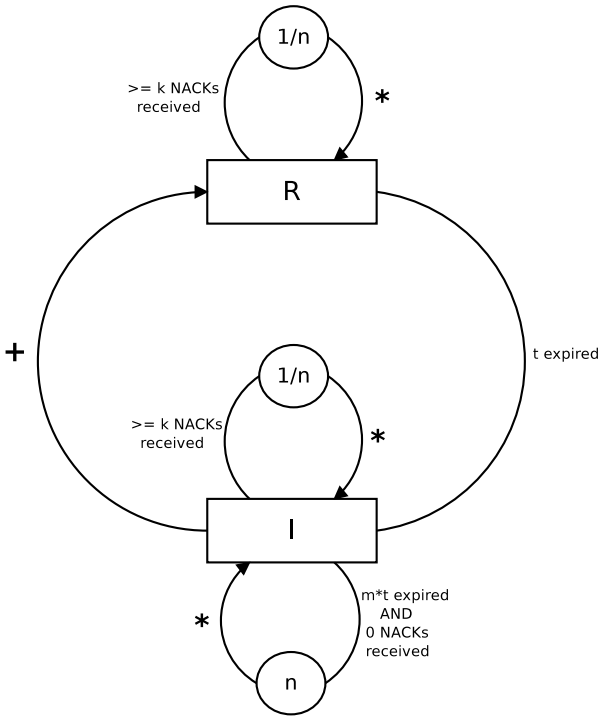


Figure 13: Send rate adaptation algorithm used by MulticastVNC.

The flow control mechanism employed by *MulticastVNC* uses a modified form of the scheme described above:

- For a rate decrease to occur, it requires a burst of k or more significant NACKs. This modification was made because during evaluation it became apparent that the original flow control scheme did not consider networks characterised by relatively high packet loss probability such as WLAN. The NACKs generated for these losses caused the transmission rate to be decreased to a much too low value.
- While the original approach used a variable timer value T, the *MulticastVNC* flow control scheme uses a constant timer value t. This change was made because the original flow control scheme performs poorly when the current send rate is relatively low or relatively high: Since the timer value T depends on the send rate as described in equation 6, the send rate is increased too slowly when its low but too fast when its rather high. Furthermore, there is no reason to let the increment timer depend on the current send rate[32].

An outline of the flow control scheme used by *MulticastVNC* is depicted in figure 13.

4.3.6 Introduction of New Message Types

The fundamental question here was whether to keep the old message types for framebuffer updates and requests and use them also for multicasting or whether to introduce new message types for the multicast case.

In case of the server-to-client multicast framebuffer update message, it was clear that a new *MulticastFramebufferUpdate* message type was needed. As discussed above, on the one hand sequence numbers had to be incorporated, on the other hand a pixel-format identifier had to be included as well.

Concerning client-to-server multicast framebuffer update requests, it would have been possible to just use traditional *FramebufferUpdateRequest* messages and interpret them as requests for multicast updates if the client registered for multicasting before. However, it was decided to introduce a new *MulticastFramebufferUpdateRequest* message because that leaves clients with the possibility to explicitly choose between asking for multicast framebuffer updates or traditional unicast framebuffer updates. Should the client detect too many lost multicast updates, it can still dynamically fall back to traditional unicast updates this way. Additionally, semantics are different with multicast update requests: while multicasting, it is not desirable to serve the needs of a single client because the purpose of sending framebuffer updates via multicast is to send them *once* for *all* connected multicast clients. Therefore, *MulticastFramebufferUpdateRequest* messages should always ask for the whole framebuffer, either as an incremental or a full

[32] It is true that more NACKs are generated at a higher send rate because more messages are sent. However, this does not mean that the send rate has to be increased faster because the number of *significant* NACKs as defined above does not change substantially.

update. In case a client wants a particular sub-region of the framebuffer, it can always resort to a traditional unicast *FramebufferUpdateRequest*.

Finally, with a NACK mechanism in place, it was necessary to introduce a new *MulticastFramebufferUpdateNACK* client-to-server message that conveys sequence numbers of messages that clients want to have retransmitted by the server.

4.3.7 Overall Resulting Design

The following section presents the final design that was derived from the solutions discussed above. The *MulticastVNC* extension of the RFB protocol introduces two new pseudo-encodings as well as two new client-to-server messages and one new server-to-client message, respectively:

MulticastVNC pseudo encoding (-831) or *IPv6MulticastVNC* pseudo encoding (-832) are used by a client to indicate that it is able to receive multicast framebuffer updates. The server responds by sending the multicast address plus port and also its multicast update interval and an identifier assigned to the client's pixel-format *and* encoding. Clients then ask for multicast framebuffer updates by issuing *MulticastFramebufferUpdateRequest* client-to-server messages (message type 242).

When the VNC server receives such a request, it does not reply immediately, instead it schedules the multicast framebuffer update to the point in time when its multicast update interval expires. This way multicast framebuffer updates are sent periodically if there are requests. Otherwise, if there are no requests, nothing is sent, saving network capacity.

If the multicast update interval has expired and there were multicast framebuffer update requests, the server sends out the update via UDP multicast. If at this point no updates were pending because the server's framebuffer did not change, it sends an empty *MulticastFramebufferUpdate* heartbeat message instead so that clients do not assume that the connection has died when there simply were no updates to send. To keep the client implementation as simple as possible, the server sends one framebuffer update for each combination of pixel-format and encoding it has to provide. Thus, each multicast framebuffer update carries a pixel-format and encoding identifier of which clients were told when sending the multicast address, as mentioned above. Furthermore, since UDP is based on datagrams with a fixed maximum size, the whole update may have to be packed into several UDP datagrams. Therefore the framebuffer contents are sent using (maybe several) *MulticastFramebufferUpdate* server-to-client messages (message type 241). These contain sequence numbers identifying the update as a whole and also the individual partial updates. By tagging updates with pixel-format and encoding identifiers, several logical data streams with different sequence numbers destined for different groups of clients can be multiplexed onto one connection.

Using these sequence numbers and the server's multicast update interval mentioned above, it is possible for clients to reorder incoming messages and detect loss of parts

of a multicast framebuffer update or of the update as a whole. They can then resort to whatever strategy they think is best. They can for instance choose to do nothing, request retransmission of missing messages by sending a *MulticastFramebufferUpdate-NACK* client-to-server message (message type 240) or request a full non-incremental multicast framebuffer update.

MulticastFramebufferUpdate messages are *the only messages sent via multicast*. Handshaking messages, client-to-server messages and all other server-to-client messages are sent using the conventional TCP unicast communication channel.

Initialization of MulticastVNC sessions and subsequent communication are described in greater detail below, including documentation of message formats and required succession of message exchange.

Session Setup

A client that understands multicast framebuffer updates tells the server so by adding the *MulticastVNC* pseudo encoding (-831) to it's *SetEncodings* message. In case the client wants to use multicast via IPV6, it adds the *IPv6MulticastVNC* pseudo encoding (-832).

If the server supports the requested feature, it tells the client about the multicast address and port to listen on for framebuffer updates and also adds its multicast update interval and a unique identifier assigned to the pixel-format and encoding the client requested.

Thus, the server sends back a *FramebufferUpdate* message consisting of one rectangle with:

- *encoding-type* set to *MulticastVNC* (or *IPv6MulticastVNC*)
- *x-position* set to the pixel-format and encoding identifier
- *y-position* set to the UDP port to listen on
- *width* set to the multicast update interval in milliseconds, which is required to be bigger than zero
- the pixel data set to an IPv4 address (or IPv6 address) in network byte order, 4 (or 16) bytes long

The client now knows about address (IPv4 or IPv6) and port to listen on for multicast framebuffer updates and sets itself up accordingly.

This way it is possible to use arbitrary user-defined ports and multicast addresses. For instance, this can be used to realize Multicast Administrative Scoping [106].

MulticastFramebufferUpdateRequest Message

After successful session setup, clients can ask for multicast framebuffer updates by sending a *MulticastFramebufferUpdateRequest* client-to-server message (message type 242). This exists so clients can explicitly ask for multicast framebuffer updates or for normal framebuffer updates via unicast TCP using *FramebufferUpdateRequest*.

Additionally, semantics are different from *FramebufferUpdateRequest*: because the purpose of sending framebuffer updates via multicast is to send them *once* for *all* connected multicast clients, it is not desirable to serve the needs of a single client. In case a client wants a particular sub-region of the framebuffer, it can always resort to a traditional *FramebufferUpdateRequest*. Therefore, a *MulticastFramebufferUpdateRequest* always asks for the whole framebuffer, with either incremental set to non-zero (true) or zero (false).

Table 6: Anatomy of a *MulticastFramebufferUpdateRequest* message.

No. of bytes	Type	[Value]	Description
1	U8	242	message-type
1	U8		incremental

MulticastFramebufferUpdate Message

Like conventional framebuffer updates, a multicast framebuffer update consists of a sequence of rectangles of pixel data. If the multicast update interval has expired and there were multicast framebuffer update requests, the server sends out the update via UDP multicast to the multicast destination it has notified the client about. If at this point no updates were pending because the server's framebuffer did not change, it instead sends an empty *MulticastFramebufferUpdate* heartbeat message containing no rectangles.

To allow any kind of multicast client and to be as flexible as possible, the server is required to keep track of which combinations of pixel-format and encoding it has to provide. For each combination, it sends out a whole multicast framebuffer update if requested by one or more clients belonging to this pixel-format and encoding group. Therefore, *MulticastFramebufferUpdate* messages have a field identifying the pixel-format and encoding of the pixel data sent. This identifier got assigned to the client's wanted pixel-format and encoding at multicast session setup. Because pixel-format and encoding are specified in the message header, all of the message's rectangles have to carry pixel data in the specified pixel-format and encoding.

Since multicast is based on UDP datagrams with a fixed maximum size, the whole update may have to be packed into several UDP datagrams. Therefore the framebuffer contents are sent using (maybe several) *MulticastFramebufferUpdate* server-to-client messages (message type 241). These contain consecutive sequence numbers identifying whole and partial updates. A whole update number identifies a logical update, i.e. the response to a *MulticastFramebufferUpdateRequest*, which may have to be split into several *MulticastFramebufferUpdate* server-to-client messages. Each of these is identified by a partial update sequence number. For each registered pixel-format and encoding combination, the server has to maintain an individual succession of sequence numbers in order to prevent clients from NACKing messages not belonging to their pixel-format and encoding group and thus to avoid useless retransmissions. This also means that clients should

not NACK messages with a pixel-format and encoding identifier different from their own. The number of rectangles is counted per each single *MulticastFramebufferUpdate* message, not per whole update.

The header is padded so that it is an exact multiple of 4 bytes to help with alignment of 32-bit pixel data.

Table 7: Header of a *MulticastFramebufferUpdate* message.

No. of bytes	Type	[Value]	Description
1	U8	241	message-type
1			padding
2	U16		id-of-pf-and-enc
4	U32		id-of-partial-update
2	U16		id-of-whole-update
2	U16		number-of-rectangles

MulticastFramebufferUpdateNACK Message

If a client notices lost messages by examining sequence numbers of received partial updates, it can choose to request retransmission of consecutive missing partial updates by sending a *MulticastFramebufferUpdateNACK* client-to-server message (with a message type of 240). This message contains the sequence number of a missing partial update which optionally indexes the start of a consecution of further partial updates. The total number of consecutive partial updates is counted including the first indexing partial update. Clients should not request retransmission of partial updates based on receipt of a *MulticastFramebufferUpdate* message tagged with a pixel-format and encoding ID different from their own in order to avoid useless retransmissions.

If the server receives such a *MulticastFramebufferUpdateNACK* message, it can choose to resend the requested partial updates or simply do nothing, clients are not guaranteed to get repair data back.

Table 8: Anatomy of a *MulticastFramebufferUpdateNACK* message.

No. of bytes	Type	[Value]	Description
1	U8	240	message-type
1	U8		padding
2	U16		number-of-partial-upds
4	U32		id-of-partial-update

5 CollabKit Implementation

This section documents the realisation of the individual implementation goals defined in section 4. What it *does not* deliver is a description of every line of code written. What it *does* provide, however, is an outline of the general implementation strategies that were used to implement new software and a summary of the way changes were made to existing code. The complete source code referenced here is available through the CollabKit project page at `http://wiki.informatik.hu-berlin.de/nomads/index.php/CollabKit` and in a Subversion repository which can be found at `https://devel-rok.informatik.hu-berlin.de/svn/magicmap/CollabKit/sources`.

5.1 Multi-User Functionality

5.1.1 VNC Server MPX Extension

The version of x11vnc that was used as a starting point was its development branch pulled from the LibVNCServer code repository [20]. The extension of this codebase with multi-pointer support basically consisted of the creation of MPX master devices for every connecting client, the assignment of custom cursors to these new master pointers and the adaptation of all client input handling to be multi-device aware. These individual parts of implementation work are documented in the next subsections.

Creation of Master Devices

The functions called by x11vnc whenever a new client connects or disconnects are `new_client()` and `client_gone()`, respectively, both located in the file `connections.c`. These were extended with calls to the functions `createMD()` and `removeMD()` which contain the low level Xlib function calls necessary to create or delete MPX master device pairs. Those two utility functions were implemented in the new file `xi2_devices.c`. For later use the numeric identifiers of the created devices are stored in the per-client data structure `clientData`.

Creation of Custom Cursors

To dynamically create custom cursors, the vector graphics library *cairo* [6] and the X11 cursor management library *libXcursor* [22] were used. Using the drawing functions provided by cairo, a coloured and labelled cursor image is rendered to off-screen memory whenever a new client connects. This cursor image is then bound to the client's newly created MPX master pointer by employing libXcursor functions. The newly written function `setPointerShape()` that does this work is implemented in the file `xi2_devices.c` as well.

Multi-Device Support in Input Handling

Pointer. Regarding pointer motion and button events, `pointer()` located in the file `pointer.c` is the central function handling input arriving from connected clients.

For button events, this function calls `update_x11_pointer_mask()` which in turn calls `do_button_mask_change()` that finally calls `XTestFakeButtonEvent_wr()` located in the file `xwrappers.c`. All these functions were extended with an additional argument, the device identifier of the master pointer belonging to the client that sent the event. Finally, in `XTestFakeButtonEvent_wr()` all calls to the legacy single-pointer `XTestFakeButtonEvent()` library function were replaced with its multi-pointer aware MPX equivalent `XTestFakeDeviceButtonEvent()`, using the passed device identifier.

When handling motion events, `pointer()` instead calls `update_x11_pointer_position()`, which in turn calls `XTestFakeMotionEvent_wr()` located in `xwrappers.c`. As with the button handling before, all function signatures were extended with an additional argument, the device identifier of the pointer belonging to the client that sent the motion event. Similarly, in `XTestFakeMotionEvent_wr()`, all calls to the single-pointer `XTestFakeMotionEvent()` were replaced with its device-aware counterpart `XTestFakeDeviceMotionEvent()`.

Keyboard. Concerning keyboard input arriving from connected clients, `keyboard()` located in the file `keyboard.c` handles all such events.

This function either calls `do_button_mask_change()`, `modifier_tweak_keyboard()`, or `XTestFakeKeyEvent_wr()` in `xwrappers.c` directly. The function `modifier _tweak_keyboard()` in turn calls either `xkb_tweakkeyboard()` or `tweak_mod()`. Again, all these functions had to be extended with an additional argument, the device identifier of the keyboard belonging to the client that triggered the event. Similar to what was done concerning pointer events, in `XTestFakeKeyEvent_wr()` all calls to `XTestFakeKeyEvent()` were replaced with calls to the multi-device function `XTestFakeDeviceKeyEvent()` that makes use of the keyboard device identifier passed along.

5.1.2 Annotation Tool MPX Extension

As outlined in subsection 4.2.2, it was decided to extend the annotation tool Gromit [15] with support for multiple pointers. The goal was to allow some remote users to *concurrently* draw graphical annotations onto the central desktop while other users keep on working normally.

For drawing onto the screen and handling of input devices, Gromit relies on the widget toolkit GTK+ [17]. Since GTK+ features MPX support in its most recent development versions, the remaining task was to change Gromit itself to be aware of multiple pointers. Like all legacy applications, Gromit assumes that there is only a single pointer and keyboard. The internal data structures and functions in the application are laid out accordingly. In order to enable drawing with multiple pointers in Gromit, those internal data structures and associated code had to be modified so that relevant data is stored *per device*.

The individual steps taken to make these modifications were:

1. Identification of variables and data structures that belong to the context of an input device.
2. Refactoring of these variables and data structures into a generic device-specific data structure, which will get instantiated dynamically *per device*.
3. Adaptation of associated code to be device aware and to handle the new per-device data structures.
4. Adaptation of code that does not deal directly with the identified data structures or variables, but nonetheless deals with input devices.

The above steps describe how Gromit specifically was modified, but probably are applicable to most other programs as well. They are documented in detail in the following subsections.

The version of Gromit used was the latest release version 20041213. Concerning GTK+, the development snapshot 2.90.1 was used.

Identification of Device-Specific Variables and Data Structures

The search for variables and data structures belonging to the context of an input device was rather trouble-free with the Gromit codebase: most variables and data structures were named after their intended use. Gromit saves all relevant data in a single global data structure named `GromitData`:

```
typedef struct {
   GtkWidget   *win, *area, *panel, *button;
   GdkCursor   *paint_cursor, *erase_cursor;
   GdkPixmap   *pixmap;
   GdkDisplay  *display;
   GdkScreen   *screen;
   gboolean     xinerama;
   GdkWindow   *root;
   gchar       *hot_keyval;
   guint        hot_keycode;
   GdkColormap *cm;
   GdkColor    *white, *black, *red;
   GromitPaintContext *default_pen;
   GromitPaintContext *default_eraser;
   GromitPaintContext *cur_context;
   GHashTable  *tool_config;
   GdkBitmap   *shape;
   GdkGC       *shape_gc;
   GdkGCValues *shape_gcv;
   GdkColor    *transparent, *opaque;
   gdouble      lastx, lasty;
   guint32      motion_time;
```

```
   GList       *coordlist;
   GdkDevice   *device;
   guint        state, timeout_id, modified, delayed, maxwidth,
                width, height, hard_grab, client, painted, hidden;
} GromitData;
```

Of the variables and pointers contained in the above data structure, `lastx`, `lasty`, `device` and `hard_grab` obviously contain information related to the mouse pointer. They therefore were first candidates for transfer into a device-specific data structure. Further study of the Gromit source code, especially of functions related to mouse pointer input, revealed some more per-device variables and data structures, completely listed in the next subsection.

Transfer into Device-Specific Data Structure

After identification, all of the device-specific variables and pointers to data structures were then transferred into a generic device-specific data structure which will get instantiated for each input device.

```
typedef struct {
   gdouble lastx;
   gdouble lasty;
   guint32 motion_time;
   GList* coordlist;
   GdkDevice* device;
   guint index;
   guint state;
   GromitPaintContext *cur_context;
   gboolean is_grabbed;
   gboolean was_grabbed;
} GromitDeviceData;
```

This `GromitDeviceData` data structure contains all variables and data structure pointers containing device-specific information plus a few new ones: `is_grabbed` and `was_grabbed` were introduced in order to be more meaningful than `hard_grab`, which was used to save more than two states. `index` is helper variable holding the associated device's number.

The transferred variables and pointers listed above were removed from the global data structure `GromitData` and replaced by the single entry

```
GHashTable *devdatatable;
```

identifying a hash table containing pointers to `GromitDeviceData` data structures as values. The corresponding hash keys are the `GdkDevice` pointers of the associated device.

Adaptation of Device-Specific Code

After all device-specific data structure pointers and variables were transferred into a separate data structure, the code using them had to be adapted.

A first step was to take care of proper allocation and de-allocation of `GromitDeviceData` data structures whenever a new input device is added or removed: Gromit gets a list of available input devices by calling `gdk_display_list_devices()` in the function `setup_input_devices()`. This was replaced by calls to the new `GdkDeviceManager` API introduced with GTK+ version 2.90. The code was further modified to allocate a `GromitDeviceData` data structure for each reported input device and insert a pointer to it into the `GromitData->devdatatable` hash table described above. The `GdkDevice` pointer associated with each input device serves as the hash key for the respective entry in `devdatatable` to later allow efficient look-up of the `GromitDeviceData` data structure corresponding to the device. When input devices are removed, the corresponding `GromitDeviceData` data structure is de-allocated as well.

In a second step the remaining code had to be adapted to access the new per-device data structures. This meant looking up the `GromitDeviceData` data structure belonging to the device in question and using the variables contained in there instead of the old global ones.

Affected were drawing functions like `paint()`, `paintto()`, `paintend()`, `gromit_draw_line()` and `gromit_draw_arrow`, but also helper functions like `gromit_select_tool()`, `gromit_coord_list_prepend()`, `gromit_coord_list_free()` and `gromit_coord_list_get_arrow_param()`. The helper functions had to be extended with an additional argument passing a device identifier along, the drawing functions already got the `GdkDevice` pointer from the GTK+ events that trigger them.

Adaptation of Code not Dealing with Identified Variables Directly

In order to complete the multi-device extension of Gromit, some other parts of its input handling code had to be adapted as well: affected was code that directly interacts with the devices exposed by the X server without using any of the device-specific variables identified above. Instead of just changing the state of core pointer or core keyboard, the code had to be modified to change *several* MPX master devices either *at once* or *individually*.

In the case of Gromit, this were functions changing the mouse pointer's state in the X server. In the original, single-pointer version of Gromit, the mouse pointer is grabbed by the application using `gdk_display_pointer_grab()` when drawing mode is activated and released by `gdk_display_pointer_ungrab()` when the user deactivates Gromit's drawing mode. When Gromit has grabbed an input device, it is the only application receiving input events for that device from the X server. Gromit uses this technique to redirect input events to its invisible overlay drawing window that would otherwise never gain focus.

To make this input device grabbing multi-pointer aware, the individual calls to the legacy single-pointer functions `gdk_display_pointer_grab()` and `gdk_display_pointer_ungrab()` were replaced with calls to the device-aware functions `gdk_device_grab()` and `gdk_device_ungrab()`. This makes it possible to grab individual mouse pointers, enabling some users to draw annotations while others can continue working normally.

Since Gromit indicates different drawing modes by changing the mouse pointer's cursor, code concerning cursor handling had to be adapted to be device-aware as well. The single-pointer version of Gromit sets the core pointer's cursor using `gdk_window_set_cursor()`. When multiple pointing devices are available, this functions sets the cursor for all of them. It therefore was replaced by `gdk_window_set_device_cursor()`.

At this point the adaptation of Gromit to make it multi-pointer aware was done. Although the individual steps documented above deal with a particular application, they also sketch a general approach of how legacy single-input-device applications can be modified to be made multi-device aware.

5.1.3 Client Application

The designated implementation goal for the client application was to provide users running different operating systems with all needed CollabKit client functionality, integrated into an easy to use graphical user interface. As mentioned in section 4.2.3, the widget toolkit chosen for this purpose is the cross-platform C++ solution wxWidgets [67] which provides *native* look and feel on all supported platforms since it uses the platform's native API instead of emulating GUI elements. Using wxWidgets, a cross-platform CollabKit client application called *MultiVNC* was implemented that enables users running Unix or Windows to:

- search for servers available in the local subnet and connect to one by simply selecting its entry in the search list, without having to enter details like IP address or port number,
- remote control the desktop exported by this server,
- and to gather statistics about the current session.

Additionally, MultiVNC provides users with controls to export their own window onto the server's desktop. This is described in greater detail in section 4.2.4. The next subsections document how the features mentioned above were implemented.

Server Discovery

Since the server application x11vnc already does service announcement using Zeroconf [82], the manifest solution was to make use of Zeroconf in the client application as well. For that purpose, the C++ Zeroconf service discovery solution wxServDisc [66] was used because it tightly integrates with wxWidgets and is cross-platform as well. Furthermore, using wxServDisc service discovery can be built directly into the application without having to rely on separate Zeroconf frameworks like Bonjour[3] or Avahi [5].

To make use of the functionality provided by wxServDisc, a wxWidgets application can either link against the wxServDisc library or has to incorporate the wxServDisc source distribution into its build system. To avoid another external dependency, MultiVNC uses the latter approach. In order for the application to be able to discover services, it

has to instantiate one or more wxServDisc objects and also has to register for *wxServDiscNOTIFY* events. Such wxWidgets events are emitted by wxServdisc objects whenever a new service is discovered or disappears. Upon receipt of such an event, MultiVNC queries the emitting wxServDisc object for results and displays them to the user in a sidebar list. An example is shown in figure 22 on page 83. The user can then simply double click one of the entries in order to connect to the corresponding server without having to know its IP address or port.

VNC Client Functionality

The fundamental VNC client functionality in MultiVNC was implemented using the LibVNCClient library that comes with LibVNCServer [21]. This library hides most of the complexities of VNC session setup and tear-down behind a simple API. It also handles communication during an active session and provides the application using it with an in-memory framebuffer representation of the remote screen.

Within MultiVNC, all LibVNCClient functionality is encapsulated in a newly implemented *VNCConn* C++ class that wraps a wxWidgets style API around the LibVNCClient C programming interface. The VNCConn implementation decouples most of the actual VNC connection handling from the application using it by implementing its own internal event loop running in a separate thread. Upon status change, certain wxWidgets events are passed to the main application. For example, the application is notified about changes to the VNCConn's framebuffer this way. It can then request the framebuffer region in question from the VNCConn instance and display it to the user. Vice versa, mouse and keyboard input received by the application is redirected to the corresponding VNCConn object accordingly, which queues it for delivery to the server. Furthermore, the VNCConn class was fit with functionality to cater for VNC's suboptimal performance on high-latency links as described in subsection 4.2.3 on page 46: instead of waiting for a server's answer after requesting a framebuffer update, a VNCConn object can optionally be switched to a mode where it continuously requests updates at a specified rate. In MultiVNC, this mode of operation is called *FastRequest*.

Taking into account the general use cases identified in section 2, an additional mode of operation was implemented as well: for those scenarios where participants are located in the same room and are able to see the presenter's screen directly, a mode of operation similar to what x2x, x2vnc, Win2VNC or Synergy [68, 94, 59, 48] implement was found to be quite valuable in terms of user experience. Within MultiVNC, this is called *seamless edge connector* mode: users can move the mouse pointer off one side of their local desktop and the mouse pointer will appear on the server's remote desktop. This is implemented by creating a small, few pixels wide window that covers one screen edge of the user's local desktop. Whenever the mouse pointer moves into this window, MultiVNC grabs mouse and keyboard and redirects all input to the remote desktop instead. When the pointer is moved back towards the opposite edge on the remote screen, it reappears on the local desktop and mouse and keyboard are released. This way it is possible to control the *remote* desktop as if it were part of a *local* multi-monitor setup. An example session is shown in figure 16 on page 78.

Gathering of Session Statistics

For proper evaluation of the implemented VNC multicast extension, the client application furthermore needed functionality to gather statistics and measure performance of the current connection. It was found that measuring achieved *throughput*, *latency* of updates and multicast *loss ratio* would be adequate.

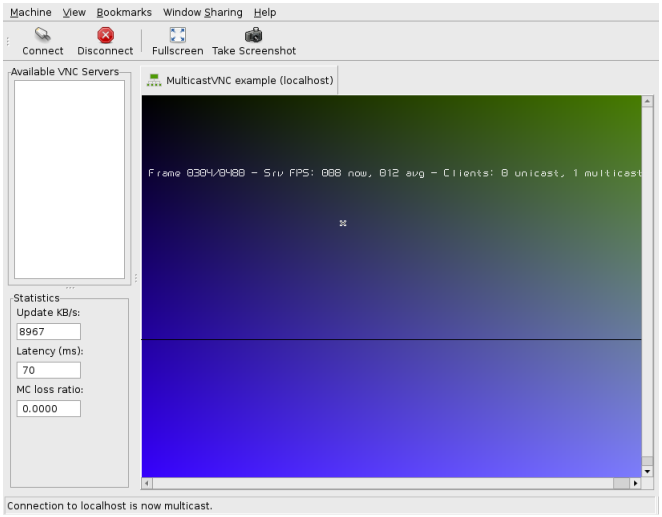


Figure 14: MultiVNC displaying statistics in the lower left while connected to a multicast test server.

To have an indicator on the throughput achieved at the client side, MultiVNC gathers per-second statistics of the number of bytes received as well as of the number of updated framebuffer bytes. There can be a difference between both numbers because VNC pixel data sent over the wire may be compressed.

To obtain latency values, two different methods are used, depending on server support: The generic method working with every VNC server requests a specific subregion of the server's framebuffer and measures the time passing till receipt of the framebuffer update containing that region. However, since VNC servers are allowed to accumulate framebuffer update requests and answer with a single update in reply to several requests [115, p. 22], this can be inaccurate. Thus, a second method of measuring server answer time that makes use of the xvp VNC extension [38] was implemented as well: Here the client sends a specifically crafted xvp message which results in the server replying with an xvp error message, whose delivery time is measured. The advantage of this method is that the server responds immediately instead of accumulating requests, its drawback is that server support is relatively scarce. It has to be noted that both methods do not necessarily give the exact data packet round trip time of the network in use. Instead, the measured values correspond to the time the server takes to craft a reply plus the round trip time. This server answer time can be considerably different from the network's RTT

when the server is busy, for instance when sending large framebuffer updates to other clients.

In order to have a useful indicator on multicast datagram loss, MultiVNC uses a moving average of length 10 over individual per-second loss ratios: Loss ratios S are calculated each second according to formula 10. The calculated value is put into a queue of maximum size 10 and the datagram counters used in formula 10 are reset to 0.

$$S = \begin{cases} \frac{lost\_datagrams}{received\_datagrams+lost\_datagrams}, & if\ received\_datagrams > 0 \\ -1, & if\ received\_datagrams = 0 \end{cases} \tag{10}$$

If there is at least one $S \geq 0$ in the current $(S_{now}, S_{now-1}, ..., S_{now-9})$ sequence, the actual multicast loss ratio R is computed each second as an average of those S in $(S_{now}, S_{now-1}, ..., S_{now-9})$ where $S \geq 0$, i.e. using a new sequence of length $m \,|\, 1 \leq m \leq 10$ containing only valid per-second loss ratios:

$$R = \frac{\sum_{k=1}^{m} S_k}{m} \tag{11}$$

MultiVNC informs the user about the current state of the connection by displaying the latest measured values in a sidebar as shown in figure 14. To allow for later analysis and visualization of session statistics data, MultiVNC can furthermore tag the individual values with absolute and relative timestamps and log them to files.

5.1.4 Client-to-Server Window Sharing

Since VNC is already used for distributing the server's screen to connected clients, it was manifest to also use it for the client-to-server window sharing functionality. In this mode of operation, the client-server roles are reversed: when a client computer exports one of its windows to the server machine's desktop, it acts as a *VNC server* whereas the actual server machine acts as a *VNC client* receiving windows. Because it is the user at the client machine who decides if and when to export a window, the roles regarding connection setup are reversed as well: in this mode, it is the VNC server application running on the client computer that initiates the connection. This mode of operation called *reverse VNC* requires specific support in both client and server software: the client application must be able to run in a *listening* mode that waits for incoming connections, the server application has to be able to connect to such a listening client.

As mentioned in section 4.2.4, x11vnc and TightVNC were chosen as the VNC server software to run on client machines. Both can connect to a listening VNC viewer and are able to share single windows instead of the whole desktop. To provide a smooth user experience, VNC server startup and window selection were integrated into MultiVNC: users can start sharing one of their windows to the server machine by simply selecting

the corresponding menu entry. The VNC server application is started in the background by MultiVNC and the user is asked to select a window to share.[33]

In order to receive windows exported by client computers, the CollabKit server machine runs a VNC viewer in listening mode. The software chosen for this purpose is an improved version of the original X11 TightVNC client software [44]. Since this viewer application forks itself whenever a new reverse connection comes in, several windows shared by different participants can be displayed on the server desktop. The individual viewer windows can be moved around freely on the server's desktop and also adapt to size changes of user's exported windows because the TightVNC viewer in use supports the DesktopSize pseudo-encoding [115, p. 43]. An example session is shown in figure 23 on page 84.

5.2 Multicast Extension of VNC

The MulticastVNC extension of the RFB protocol described in section 4.3 was implemented atop the LibVNCServer library [21]. This was chosen as a codebase to build upon because it is the only general-purpose VNC library around. Besides, both the original x11vnc and its multi-pointer version use LibVNCServer internally as well.

The following subsections document how the LibVNCServer codebase was extended in order to implement the MulticastVNC protocol extension. The code used as a starting point was pulled directly from the LibVNCServer development repository [20].

5.2.1 Declaration of Message Types

A first step was to declare the new message types introduced with MulticastVNC in the LibVNCServer header files. This was done analogous to how existing message types are declared: C struct declarations and size defines were added to the `rfb/rfbproto.h`[34] file for each of the new message types.

Therefore, the *MulticastFramebufferUpdateRequest* client-to-server message drafted in section 4.3 was declared as follows:

```
typedef struct {
   uint8_t type;  /* always rfbMulticastFramebufferUpdateRequest */
   uint8_t incremental;
} rfbMulticastFramebufferUpdateRequestMsg;
#define sz_rfbMulticastFramebufferUpdateRequestMsg 2
```

[33] x11vnc comes with its own selection mechanism, the TightVNC executable has to be supplied with a window identifier on startup. Because the original Windows version 1.3.9 of TightVNC requires an already running server instance in order to accept command-line arguments, the version bundled with MultiVNC was slightly modified to allow instant startup with command-line arguments.

[34] In the following, all file paths mentioned are relative to the root directory of LibVNCServer's code repository.

Similarly, this code was added to declare the C data type representing a *MulticastFramebufferUpdate* server-to-client message:

```
typedef struct {
   uint8_t type;  /* always rfbMulticastFramebufferUpdate */
   uint8_t pad;
   uint16_t idPixelformatEnc; /* pixelformat and encoding id */
   uint32_t idPartialUpd;  /* id of this partial update */
   uint16_t idWholeUpd;    /* id of the update as a whole */
   uint16_t nRects;        /* number of rectangles per message */
   /* followed by nRects rectangles */
} rfbMulticastFramebufferUpdateMsg;
#define sz_rfbMulticastFramebufferUpdateMsg 12
```

Finally, the C struct representing a *MulticastFramebufferUpdateNACK* client-to-server message was coded as below:

```
typedef struct {
   uint8_t type;  /* always rfbMulticastFramebufferUpdateNACK */
   uint8_t pad;
   uint16_t nPartialUpds;  /* number of missing partial updates */
   uint32_t idPartialUpd;  /* id of first missing partial update */
} rfbMulticastFramebufferUpdateNACKMsg;
#define sz_rfbMulticastFramebufferUpdateNACKMsg 8
```

After the C structs representing the individual message types were added, aliases for message type and pseudo-encoding constants were added to make other code more readable and less error-prone:

```
#define rfbMulticastFramebufferUpdate 241
#define rfbMulticastFramebufferUpdateRequest 242
#define rfbMulticastFramebufferUpdateNACK 240
#define rfbEncodingMulticastVNC       0xFFFFFCC1
#define rfbEncodingIPv6MulticastVNC   0xFFFFFCC0
```

The last step needed to make the newly declared MulticastVNC message types usable within LibVNCServer was to add them to the general abstracting `rfbClientToServerMsg` and `rfbServerToClientMsg` C unions.

5.2.2 Implementation of Session Setup

After message types and pseudo-encodings used by the MulticastVNC extension were added to the appropriate LibVNCServer header file, the session setup mechanism described on page 57 could be implemented.

At the server side, some new members were added to the `rfbScreenInfo` struct declared in `rfb/rfb.h`: variables to hold multicast destination address and port as well

as multicast time-to-live were appended at the end of this C struct representing a VNC server instance. If these are set at server startup, the newly introduced `rfbCreateMulticastSocket()` function implemented in `libvncserver/sockets.c` is called by `rfbInitSockets()`. Depending on the multicast address given, `rfbCreateMulticastSocket()` either creates an IPv4 or IPv6 datagram socket, sets the multicast TTL and connects the socket to the specified multicast destination address. It also saves multicast address and port in a `sockaddr_storage` variable added to `rfbScreenInfo` for later reference.

As stated by the session setup specification on page 57, a client connecting to a MulticastVNC server will announce its possible multicast support by adding the `rfbEncodingMulticastVNC` or `rfbEncodingIPv6MulticastVNC` pseudo-encodings to it's *SetEncodings* message. In the LibVNCServer sources, this could simply be achieved by adding both to the encodings requested by the `SetFormatAndEncodings()` function defined in `libvncclient/rfbproto.c`.

When received by the server, such an RFB *SetEncodings* message is handled by `rfbProcessClientNormalMessage()` located in `libvncserver/rfbserver.c`. If the MulticastVNC pseudo-encoding the client requested matches the socket type created by `rfbCreateMulticastSocket()` at server startup, the `enableMulticastVNC` flag for that client is set, which in turn triggers its setup for MulticastVNC: The pixel-format and encoding identifier is found by comparing the client's requested pixel-format and encoding with that of other already registered clients. If they differ, a new pixel-format and encoding identifier is assigned to that client. The client's pixel-format and encoding also decide if it becomes part of an already existing pixel-format and encoding group and shares the groups sent buffer (described in section 5.2.4) and other resources or if those resources have to be newly allocated. Following this setup, `rfbSendMulticastVNCSessionInfo()` gets called. This function, newly implemented in `libvncserver/rfbserver.c`, sends the server's multicast session data to the client by crafting a *FramebufferUpdate* message as described on page 57: the rectangle's encoding is set according to the server's multicast socket address type, multicast destination IP address and port are taken from the `sockaddr_storage` saved by `rfbCreateMulticastSocket()`.

Back at the client side, the received multicast session data is handled in `HandleRFBServerMessage()`, located in `libvncclient/rfbproto.c`: when this function encounters an RFB rectangle with it's encoding set to one of the MulticastVNC pseudo-encodings, it checks the received multicast address for validity and calls `CreateMulticastSocket()`, newly implemented in `libvncclient/sockets.c`. According to the received multicast session data, this function creates a datagram socket with the right address type, binds it to the client's address and joins the specified multicast group. It also increases the socket's receive buffer size if possible to avoid datagrams being discarded when the socket is not read frequently enough. Finally, the client's receive buffer is allocated by calling `packetBufCreate()`. The receive buffer is implemented in `libvncclient/packetbuf.h` and `libvncclient/packetbuf.c`.

With this step, MulticastVNC session setup is complete at both the server and the client side, MulticastVNC messages can be sent and received.

5.2.3 Implementation of Message Handling

After the MulticastVNC session is setup is complete, clients can ask for multicast framebuffer updates by sending *MulticastFramebufferUpdateRequest* messages to the server. For this purpose, a `SendMulticastFramebufferUpdateRequest()` function was implemented in `libvncclient/rfbproto.c`. This function simply crafts a *MulticastFramebufferUpdateRequest* message and sends it to the server via the ordinary TCP unicast communication channel.

As described on page 56, a VNC server receiving such a request does not reply immediately. Instead, it schedules the multicast framebuffer update to the point in time when its multicast update interval expires. Since a MulticastVNC server has to send a multicast framebuffer update for each combination of pixel-format and encoding that was requested, all clients belonging to the same pixel-format and encoding group share a common »update pending« flag. This is implemented as one boolean value allocated on the heap with associated clients maintaining pointers to it. Whenever a client requests a multicast framebuffer update, the shared flag pointed to by `multicastUpdPendingPtr` is set by `rfbProcessClientNormalMessage()` in `libvncserver/rfbserver.c`. When another client with the same pixel-format and encoding requests a multicast framebuffer update, the same flag gets set, ensuring that a multicast framebuffer update is just sent once for each requested combination of pixel-format and encoding.

After the server's multicast update interval has expired, `rfbProcessEvents()` in `libvncserver/main.c` iterates over the list of connected clients, checking if the flag pointed to by `multicastUpdPendingPtr` is set. If it is, a framebuffer update with this pixel-format and encoding is multicasted and the shared flag unset. This way, a multicast framebuffer update is sent *once* for each combination of pixel-format and encoding requested.

The actual dispatch of multicast framebuffer updates is done by `rfbSendMulticastFramebufferUpdate()`, which was added to `libvncserver/rfbserver.c`. This function either multicasts the *modified* parts of the server's framebuffer in the specified pixel-format and encoding[35] or sends an empty *MulticastFramebufferUpdate* heartbeat message[36] so that clients do not assume that the connection has died when there were simply no updates to send. When there are modified parts of the framebuffer, they are marked within LibVNCServer using a certain `sraRegion` data type which basically constitutes a set of coordinates representing several rectangular subareas of a server's

[35] At the time of writing, only *Raw* [115, p. 33] and *Ultra* encoding are implemented.

[36] This is true when the *incremental* flag of the corresponding *MulticastFramebufferUpdateRequest* was set. If *incremental* was set to false in at least one request, the *complete* framebuffer contents are transmitted for this pixel-format and encoding combination even if no change to the framebuffer occurred.

in-memory screen. Such an `sraRegion` can be queried for all the individual rectangles it consists of. Thus, `rfbSendMulticastFramebufferUpdate()` basically takes a list of rectangles, a pixel-format and an RFB encoding as its arguments. The fundamental issue here was that the individual subareas can be so big that the pixel data they represent does not fit into a single UDP datagram. Therefore, over-sized rectangles needed to be split up, resulting in one logical multicast framebuffer update being sent using several individual *MulticastFramebufferUpdate* messages. The algorithm used for crafting individual partial updates out of one whole update is shown in figure 15.

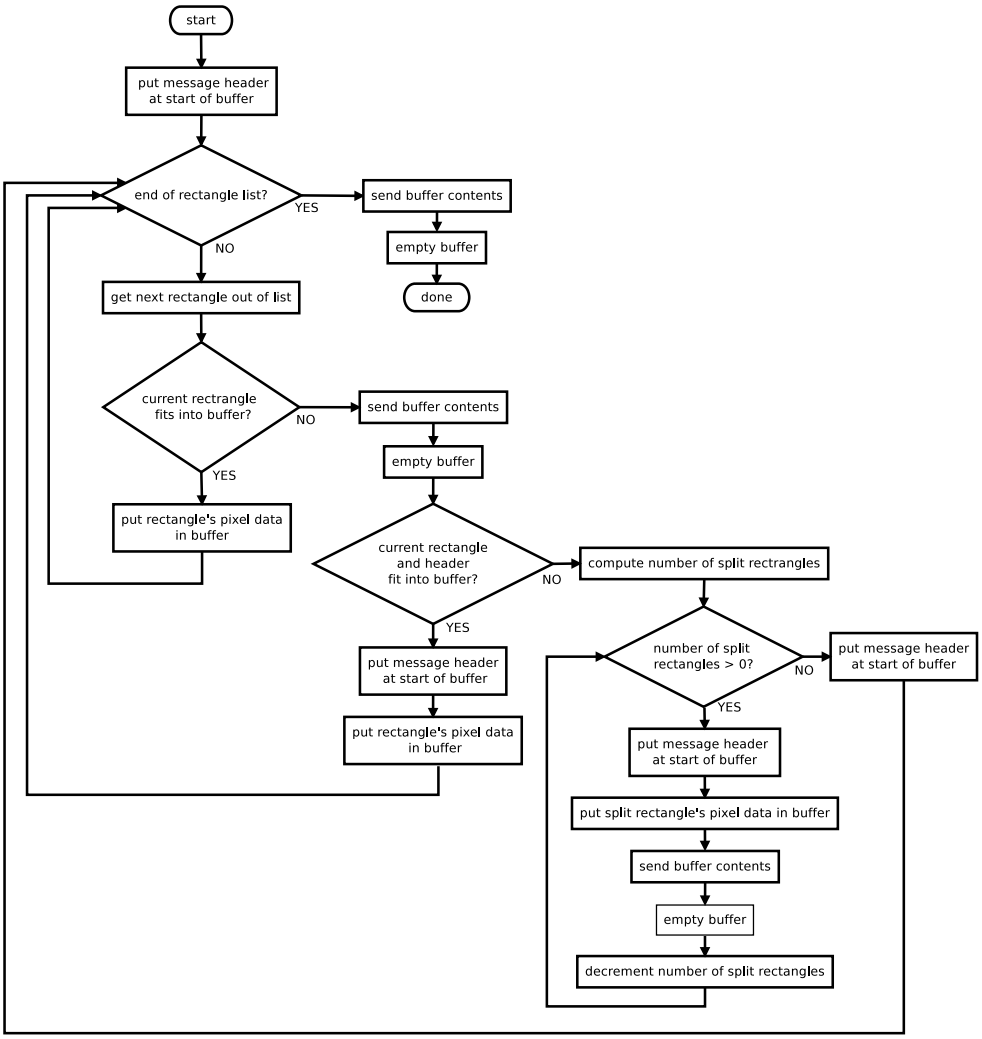


Figure 15: The algorithm used within `rfbSendMulticastFramebufferUpdate()` depicted as a flowchart. The buffer is sent via `rfbWriteExactMulticast()` implemented in `libvncserver/sockets.c`, which writes to the multicast socket created on server startup.

For each complete multicast framebuffer update, `rfbSendMulticastFramebufferUpdate()` labels the individual *MulticastFramebufferUpdate* messages with the same *id-of-whole-update*. Each single message gets tagged with its own *id-of-partial-update* in ascending order. To make client-side detection of lost messages work, the server maintains an individual succession of sequence numbers for each registered pixel-format and encoding group, implemented as shared values allocated on the heap. Using these sequence numbers and the server's multicast update interval together with possible heartbeat messages, clients are able to detect lost messages as well as failure of connection.

A big part of the needed client-side functionality was implemented in `rfbProcessServerMessage()` which was added to `libvncclient/vncviewer.c`. This function is to be called from a client application's main loop and takes care of requesting multicast framebuffer updates, handles messages and deals with poor quality connections: in case no multicast messages at all arrive within a certain time span, it falls back to traditional unicast VNC. It does not take the measured multicast message loss ratio into consideration, actions upon high loss ratio are intentionally left to the client application using the library.

For low level message handling, `rfbProcessServerMessage()` periodically requests multicast framebuffer updates and checks whether new messages have arrived at the client's multicast socket or if there still are messages in the receive buffer by invoking `WaitForMessage()`. If there are unread messages, `HandleRFBServerMessage()` located in `libvncclient/rfbproto.c` is called to process them. It reads in messages via `ReadFromRFBServerMulticast()` which was added to `libvncclient/sockets.c`. This function always first tries to read as many packets as possible from the socket into the client's receive buffer and then returns the requested number of bytes. A received *MulticastFramebufferUpdate* message is only handled if it is tagged with the client's pixel-format and encoding, otherwise it is discarded. When the message was accepted, `HandleRFBServerMessage()` first calculates the number of lost messages by looking at the sequence numbers of received partial updates and then passes the update's individual rectangles to the corresponding VNC encoding handler which decodes the rectangle and puts it into the client's framebuffer. At the time of writing, only *Raw* and *Ultra* encoding [115, p. 33] are implemented for MulticastVNC, but others can be added by modifying the corresponding handler functions.

5.2.4 Implementation of the NACK mechanism

In case partial updates are found to be missing, `HandleRFBServerMessage()` requests their retransmission by calling `SendMulticastFramebufferUpdateNACK()`. This function, implemented in `libvncclient/rfbproto.c`, sends a *MulticastFramebufferUpdateNACK* client-to-server message that contains the sequence number of *one* missing update or a sequence number *range* of several consecutive missing updates.

At the server side, it is of course necessary to keep track of which contents were sent with which message for the NACK mechanism to work. A common approach is to simply

save sent messages for a certain time span to be able to resend them when requested. While this solution works reasonably well, it also is unnecessarily memory-intensive: it needlessly uses extra memory to store data that is already present in the server's framebuffer. Therefore, a better approach is to just store coordinates of the sent framebuffer subregions instead of storing the pixel data itself. For this purpose, a ring buffer of data structures associating sequence numbers with framebuffer subregions was implemented in the files `libvncserver/partialupdateregionbuf.h` and `libvncserver/partialupdateregionbuf.c`. Again, all clients with the same pixel-format and encoding share a common buffer on the heap which they access via pointers. The individual `partialUpdRegion` ring buffer elements look like:

```
typedef struct _partialUpdRegion {
   uint16_t idWhole;
   uint32_t idPartial;
   sraRegionPtr region;
   rfbBool pending;
   uint32_t sendrate;
   rfbBool sendrate_decreased;
} partialUpdRegion;
```

As the name implies, they relate the sequence numbers of individual partial updates or *MulticastFramebufferUpdate* messages to the framebuffer subregions that were sent with the message in question. Furthermore, the `partialUpdRegion` struct contains a member that indicates whether that particular update was NACK'ed by a client or not.

Thus, when a *MulticastFramebufferUpdateNACK* is request received, this message gets handled by `rfbProcessClientNormalMessage()` in `libvncserver/rfbserver.c`. This function checks whether or not the sequence numbers of the reported lost partial updates are contained in the shared ring buffer mentioned above. If they are, they get marked as requested by setting the corresponding `partialUpdRegion`'s `pending` member to `TRUE`. The shared ring buffers are later checked by `rfbProcessEvents()` in `libvncserver/main.c` by calling `rfbSendMulticastRepairUpdate()` which checks whether repair data needs to be sent. If a shared buffer is marked as dirty, this function checks it for partial updates marked as requested, looks up the associated sequence numbers and framebuffer subregion and multicasts *MulticastFramebufferUpdate* repair messages with the specified sequences numbers and contents.

5.2.5 Implementation of Multicast Flow Control

The implemented flow control scheme is the one described in section 4.3.5. The variables R and I mentioned there map to the `rfbScreen` struct's member variables `multicastMaxSendRate` and `multicastMaxSendRateIncrement`. The parameters k, m, n and t (figure 13 on page 54) map to the constants `MULTICAST_MAXSENDRATE_NACKS_REQUIRED`, `MULTICAST_MAXSENDRATE_INCREMENT_UP_AFTER`, `MULTICAST_MAXSENDRATE_CHANGE_FAC-`

TOR and MULTICAST_MAXSENDRATE_INCREMENT_INTERVAL, all defined in rfb/rfb.h. Start values for maximum send rate and increment are also defined there.

Rate Limiter

The first thing needed for working flow control was a mechanism that would prevent the server from sending at a higher rate than multicastMaxSendRate, i.e. a rate limiter had to be implemented. The LibVNCServer MulticastVNC implementation uses a simple token bucket algorithm in rfbWriteExactMulticast() to limit the maximum send rate: One token is interpreted as one byte which the server is allowed to send. This send credit initially is at multicastMaxSendRate and gets depleted when messages are sent out. Likewise, it is replenished in the same function depending on how much time has passed since the last call to rfbWriteExactMulticast().

Rate Increasing

To implement the rate increasing part of the MulticastVNC flow control scheme, rfbWriteExactMulticast() in libvncserver/sockets.c was extended to add the rate increment to the current maximum send rate on timer expiration and also increase the increment itself as described in section 4.3.5. Since a VNC server is not constantly transmitting data at the highest possible rate, the rate is only increased when the server actually is sending at the maximum rate, i.e. when the send credit is depleted. If behaviour was not like that, the maximum send rate would constantly be increased even though only a few bytes were sent with each call to rfbWriteExactMulticast(), resulting in much too high multicastMaxSendRate. The reason is that when the server is sending at a low rate, no (or too few) NACKs are generated so there would be no send rate decrease at all.

Rate Decreasing

A send rate decrease is considered when a *MulticastFramebufferUpdateNACK* message requesting equal to or more than MULTICAST_MAXSENDRATE_NACKS_REQUIRED partial updates is received. Like all client input, such messages are handled by rfbProcessClientNormalMessage() in libvncserver/rfbserver.c. This function checks whether or not the sequence numbers of the reported lost partial updates are contained in the shared ring buffer described in section 5.2.4 on page 74. If they are, the sendrate member of the corresponding partialUpdRegion buffer element is compared with the current maximum send rate. This sendrate variable contains the maximum send rate logged at the time the NACKed partial update was sent. If the *current* value of multicastMaxSendRate is bigger than or equal to the logged send rate, a rate decrease may occur. However, this only happens if this logged send rate has not been decreased before, indicated by the sendrate_decreased member of the buffer element. The actual send rate decrease then modifies multicastMaxSendRate and multicastMaxSendRateIncrement as described in section 4.3.5 and also sets the sendrate_decreased member of all buffer elements containing the same logged send rate.

5.2.6 Use with LibVNCServer

This subsection describes how the MulticastVNC functionality added to LibVNCServer and LibVNCClient can be used by application code:

For a server application, MulticastVNC can be enabled by setting the `multicastVNC` member of a server's `rfbScreenInfoPtr` to `TRUE`. Multicast destination address, port and TTL can be set via the `multicastAddr`, `multicastPort` and `multicastTTL` member variables. The defaults are 224.0.42.138, 5900 and 1. By setting `multicastDeferUpdateTime`, the multicast update interval can be specified.

To enable MulticastVNC for a client, the `canHandleMulticastVNC` member of the corresponding `rfbClient*` has to be set to `TRUE`. The most simple way to get a working client application is to call `rfbProcessServerMsgs()` in its main loop. This includes requesting multicast framebuffer updates, waiting for messages and handling them. It also includes some default actions regarding packet loss and timeouts as mentioned above. It is, however, also possible to not use `rfbProcessServerMsgs()` at all and to call all the functions it uses directly, thus implementing own message handling logic.

6 CollabKit Evaluation

6.1 Evaluation of Multi-User Functionality

This section documents how well CollabKit's multi-user functionality meets the requirements identified in section 2.2 on page 7. The multi-pointer extensions of x11vnc and Gromit and the CollabKit client application MultiVNC are examined in this regard.

6.1.1 Concurrent Multi-User View and Control

The ability for participants to view and control the shared remote desktop is the most fundamental requirement of all the real-time collaboration use cases presented in section 2: for the presentation scenario, only remote control facilities are needed, but for electronic classroom and professional collaboration use cases the ability to also view the shared desktop is mandatory.

The CollabKit client application MultiVNC provides appropriate remote control and view features for each of the considered use cases: for electronic classroom and professional collaboration scenarios where users may be located in different places, it supports full view and remote control capabilities, as can be seen in figure 16.

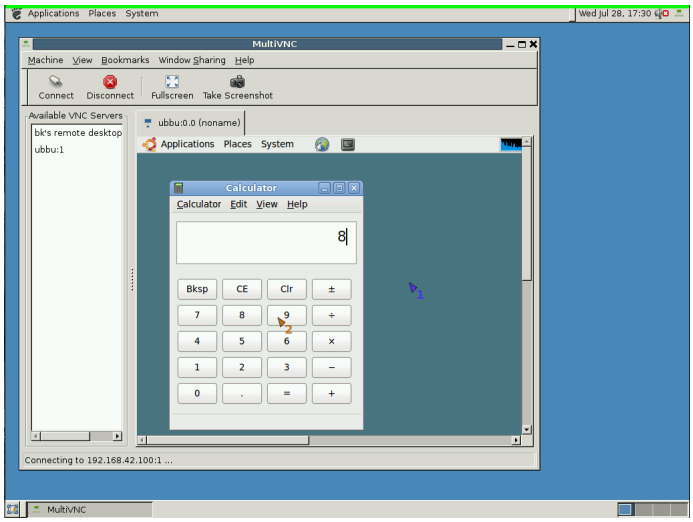


Figure 16: MultiVNC connected to a CollabKit server with *seamless edge connector mode* enabled at the northern edge of the local screen.

For presentation scenarios and all other use cases where participants are located in the same room and are able to see the shared screen directly, MultiVNC additionally provides a so-called *seamless edge connector* mode where users can move the mouse pointer off one side of their local desktop to make it appear on the server's remote desktop. When

the pointer is moved back towards the opposite edge on the remote screen, it reappears on the user's local desktop. This unobtrusive and intuitive kind of remote control makes it possible to operate the *remote* desktop as if it were part of a *local* multi-monitor setup. The edge connector is the green window that can be seen in the upper part of figure 16.

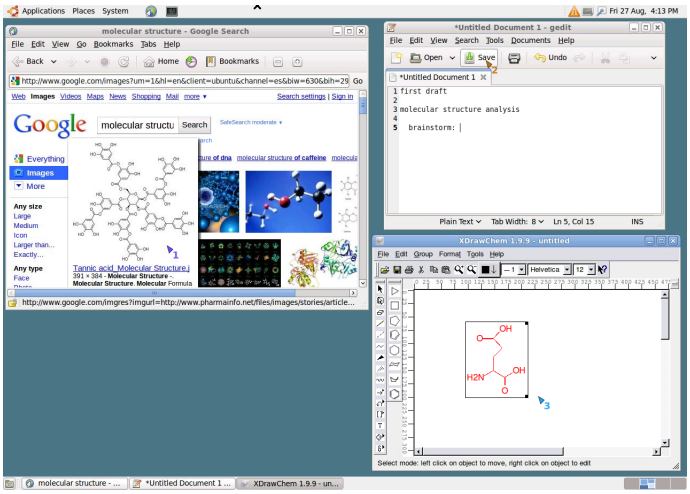


Figure 17: A desktop exported by the modified multi-pointer version of x11vnc, showing three clients each operating a different application at the same time. Each client has its own pointer with a distinctly coloured and labelled mouse cursor.

Multi-user operation was first and foremost demanded by the professional collaboration use case, but the other scenarios benefit from this as well. Unlike other systems (examined in section 3) which only provide turn-taking, CollabKit features *concurrent* multi-user remote control of a shared desktop. The modified x11vnc server used in CollabKit provides every participant with their *own* independent mouse cursor and keyboard focus, allowing users to interact with objects on the server's desktop *jointly* and *simultaneously*.

It is important to state though that at the time of writing legacy applications can only be operated flawlessly by *one* user at a time, concurrent interaction on the same desktop is only possible with *different* legacy applications. However, new applications can be designed with multi-device control in mind (like the simple shared scribble sheet depicted in figure 18 on the following page) and existing legacy applications can be modified to be made multi-device aware. In the simplest case, it may be sufficient to just link against a multi-device aware version of the underlying widget toolkit, such as GTK+ 3.0 [18].

Security Considerations

Access control, as stated by the requirements analysis section 2.2.2, was found to be equally important for all considered use cases and in fact mandatory in most situations:

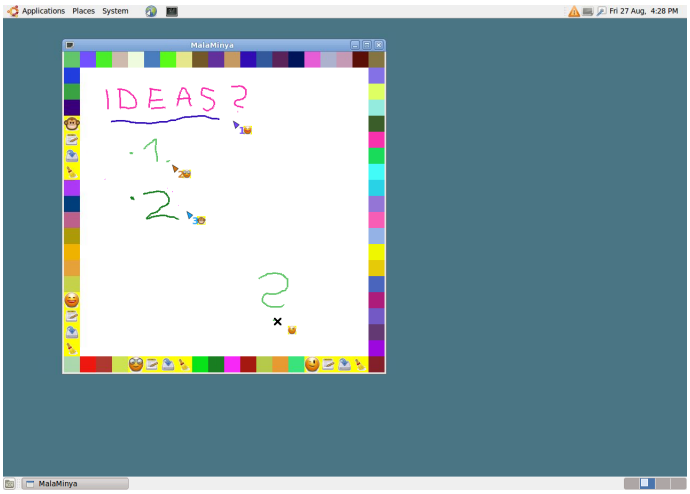


Figure 18: One local and three remote participants *concurrently* operating a simple multi-device aware scribble sheet on the shared desktop.

Depending on configuration, the modified x11vnc server used within CollabKit either simply allows everyone to connect, asks joining participants for a shared password or authenticates them using Unix user name and password. This way, unwanted users can be prevented from accessing the shared CollabKit desktop. Using this password mechanism, it is also possible to distinguish between users that are allowed to actually *operate* the shared desktop and users that can only *view* what is going on. The client-to-server window sharing feature has no such password authentication support, instead it simply pops up a window on the shared desktop asking if the incoming connection should be accepted or not. While this is certainly not an ideal solution, it works reasonably well.

As furthermore stated in section 2.2.2, confidentiality, availability and integrity of occurring communication is not imperatively needed in private and secured local area networks but becomes mandatory when data is transmitted over untrusted networks like the Internet. This essentially means that in addition to an authentication mechanism, encryption of transmitted data is needed to prevent eavesdropping or spoofing. The x11vnc server application used for CollabKit does provide encryption in form of the VeNCrypt[57] and ANONTLS VNC extensions, as does the client application MultiVNC[37].

6.1.2 Multi-User Graphical Annotations

The ability to draw graphical annotations onto the shared desktop was found to be an important functional requirement for all considered use cases. Be it in presenta-

[37] This is only supported for *unicast* connections. At the time of writing, all multicast framebuffer updates are sent unencrypted.

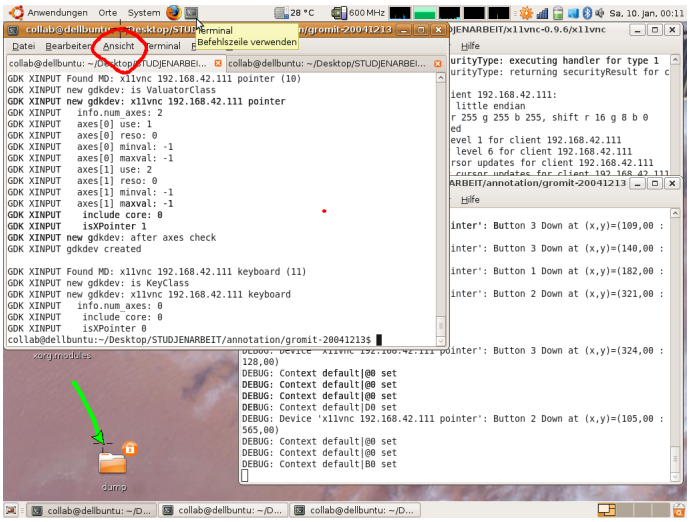


Figure 19: Two users drawing annotations onto the screen with the device-aware version of Gromit while a third one keeps operating the desktop.

tions, teaching or in a professional collaboration scenario: it is beneficial for mutual understanding when participants are able to highlight certain regions of the desktop for others. This could be either to explain something more clearly or to be able to ask more specifically about something on the shared desktop.

Within CollabKit, the use of the modified Gromit version devised in subsection 4.2.2 facilitates a *multi-user* annotation mode: a user connected to a CollabKit server can enable annotation mode by pressing the *pause* key. This activates on-screen drawing *only* for this client, others are still able to operate the shared desktop as usual, as can be seen in figure 19. While in annotation mode, the user can draw anywhere on the screen by pressing the left mouse button and can erase annotations using the right mouse button. Hitting the pause key again deactivates annotation mode for that user.

6.1.3 Cross-Platform Client

To allow for a user base as broad as possible, the CollabKit client application was required to be cross-platform or at least to be available for the different operating systems that CollabKit users are likely to run on their client computers. This essentially meant that the client application MultiVNC needs to work on the major operating systems Linux, Mac OS X and Windows.

For this purpose, the cross-platform C++ widget toolkit wxWidgets [67] was chosen because of two reasons: first, a toolkit with C bindings was needed since the underlying LibVNCClient library is written in that language. Second, out of the available widget toolkits with C bindings, wxWidgets is the only one to provide *native* look and feel on

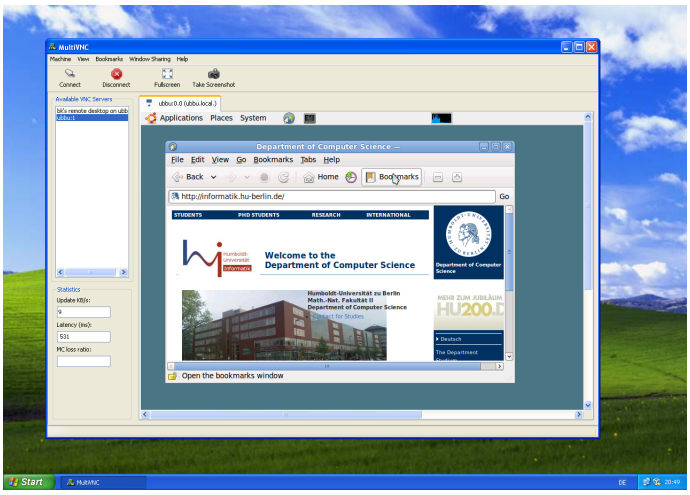


Figure 20: MultiVNC running on Windows connected to a Linux VNC server.

all supported platforms since it uses each platform's native API instead of emulating GUI elements.

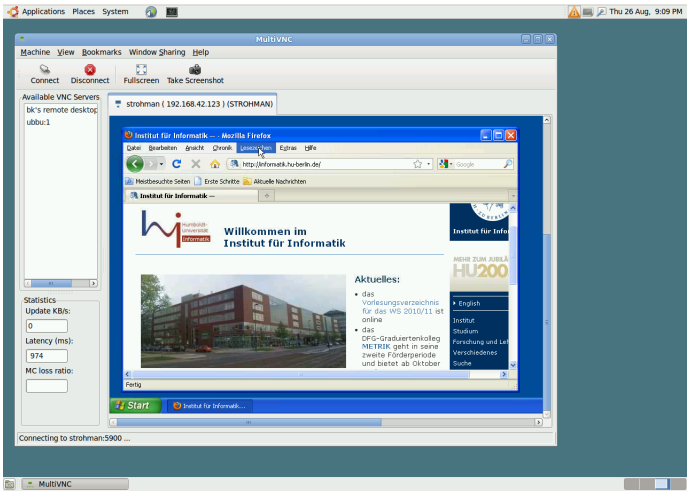


Figure 21: MultiVNC running on Linux connected to a Windows VNC server.

The posed requirement of being able to run the CollabKit client application on major operating systems could mostly be achieved: there are MultiVNC packages for most major Linux distributions and there also is a Windows installer available. Figure 20 shows MultiVNC running on Windows whereas figure 21 depicts the Linux version. The MultiVNC variant for Mac OS X is able to start up as well, but it has some problems that need to be investigated and fixed in order to make it functional.

Automatic Server Discovery

It was found in the requirements analysis in section 2.2 that one the most significant issues hampering ease of use is recurring unnecessary configuration. In the case of the client-server system CollabKit, this meant not requiring users to know of IP addresses and ports in order to connect to the CollabKit server.

Instead, the CollabKit client application was required to do automatic server discovery for the user. How it does this is described in section 5.1.3. The expected user experience is to be able to connect to a server by simply choosing one entry out of a list.

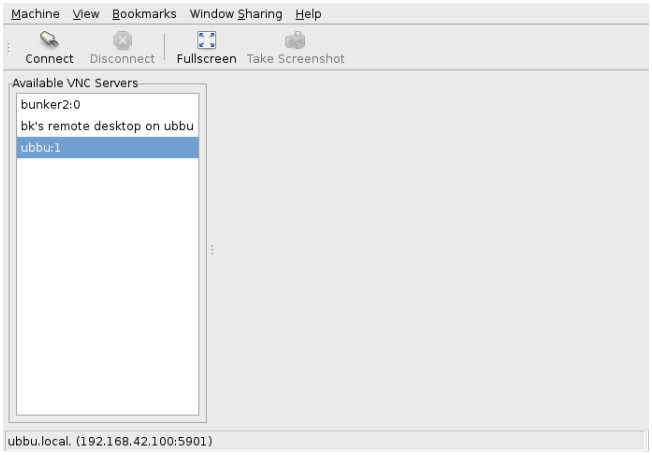


Figure 22: MultiVNC showing discovered servers in a sidebar to the left.

The CollabKit client application MultiVNC facilitates exactly this: on startup, it looks for servers advertising themselves via Zeroconf and displays the results to the user, as shown in figure 22: the user is provided with a sidebar list of discovered servers and can connect to one by simply double clicking the corresponding entry without having to know the server's IP address or port.

6.1.4 Client-to-Server Window Sharing

As stated in the requirements analysis in subsection 2.2, client-to-server window sharing is a functional requirement to the system in use posed by all three of the considered use cases: for presentations, functionality to show their own window on the presenter's desktop enables attendees to show a certain document or application to other participants. For students in an electronic classroom scenario, this is a useful feature as well since it enables them to show their local application windows to the instructor in order to get help on a specific problem. Finally, client-to-server window sharing is a fundamental feature for professional collaboration use cases because it is very likely for such computer supported collaborative work scenarios that documents or running applications need to be shown to co-workers while brainstorming and collecting information.

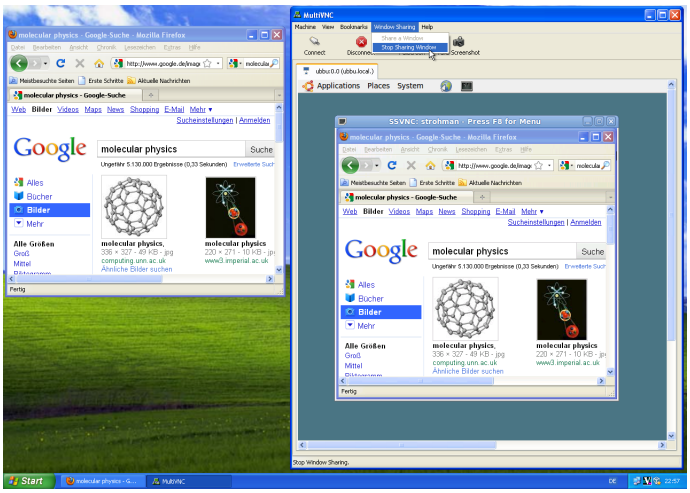


Figure 23: A session where the browser window on the left of the local Windows desktop is exported to the remote CollabKit desktop. The user is just about to end window sharing again.

When connecting to a CollabKit server using MultiVNC, the client-to-server window sharing functionality is easily accessible from within the client application's user interface: after having established a connection to the server, the user may select the »share window« entry out of the »window sharing« menu. When running Windows, a dialog window will pop up asking the user for the name of the window to share. Under Unix, it is possible to select a window by simply clicking on it. The selected window will then appear on the shared desktop, as shown in figure 23. It can be freely dragged around on the shared desktop and also be operated by other participants. When the original local window is re-sized, the exported window will be adapted in size as well.

To end window sharing, the user has to select the appropriate entry in the window sharing menu of MultiVNC. While it would technically be possible to share more than one window at a time, this is not implemented in MultiVNC at the time of writing.

Security Considerations

In terms of access control, CollabKits client-to-server window sharing feature has no password authentication support like the fundamental view and control features discussed in section 6.1.1. Instead, it simply pops up a window on the shared desktop asking if the incoming connection should be accepted or not. While this is certainly not an ideal solution, it works reasonably well.

Concerning confidentiality, availability and integrity of window-sharing data exchange, the tools used for client-to-server window sharing do support encryption via either SSH or SSL tunneling. However, SSH tunneling requires the connecting user to have an account at the server. SSL tunneling works without preparatory setup, but is vulnerable

for man-in-the-middle attacks when the server's certificate is not checked. More secure SSL tunneling is possible, but requires that the certificate of the VNC server exporting the user's window is known to the listening VNC viewer at the CollabKit server machine.

6.2 Evaluation of the MulticastVNC Extension

This section documents how well MulticastVNC performs compared to traditional unicast VNC and to what extent the predicted results regarding throughput and latency are met.

In order to make well-founded statements on unicast versus multicast performance, extensive real-world tests with a total of eight computers were carried out. Up to seven client machines were employed, measuring throughput and latency as well as MulticastVNC NACK and loss ratios using facilities built into the MultiVNC client. CPU load on the server machine was measured for some tests as well, using the `vmstat` system monitoring tool.

Figure 24: Client machines carrying out performance tests using the MultiVNC application.

The tests were conducted on a EeePC 701 4G with a 630 MHz CPU and 512 MB RAM running Ubuntu Linux 10.10 as the server machine. The computers used as client machines were quite diverse: To measure performance in a WLAN, 3 laptops with built-in wireless cards and 1 PC using a USB WLAN adapter were employed. The used machines were a JVC MP-XP 731 laptop with a 1 GHz CPU and 512 MB RAM running Debian Linux 6.0, a Lenovo X61 laptop equipped with a dual-core 1.6 GHz CPU and 3GB RAM running Ubuntu Linux 10.04, a Dell Latitude D500 laptop (1.3 GHz CPU, 512 MB RAM) running Ubuntu Linux 10.10 and a PC with a 3GHz hyper-threading CPU and 1 GB of RAM running Windows XP. Three more such PCs were available as well, but due to lack of WLAN adapters could only be used as additional client machines for tests in a LAN.

In sum, there were client 4 machines available for WLAN tests while for measurements in a LAN all 7 computers could be used as clients. Figure 24 shows the test bed in action.

A single test unit was defined to last exactly 3 minutes, resulting in 180 samples taken by each participating MultiVNC client instance. In order to put load on the clients, the server machine in all tests constantly sent 640x480 pixels of 32-bit image data with a desired frame rate of 15 frames per second. To make unicast and multicast comparable, the server was configured to delay unicast updates for the same time span as the multicast update interval, which was set to 10 milliseconds in most test cases. The VNC encoding used for the majority of tests was Raw encoding. Ultra encoding as the default was considered as well, but ultimately dismissed because with the relatively weak server machine the achievable throughput was found to be CPU bound instead of being limited by network characteristics and method of data transmission.

6.2.1 Throughput Properties

To be able to compare the throughput characteristics of VNC and MulticastVNC on a sound basis, experiments were carried out that measured throughput as seen by clients in different configurations: This included varying the number of connected clients, testing in different network environments (Fast Ethernet LAN and 802.11b WLAN) and of course changing between traditional unicast VNC and MulticastVNC.

The basic methodology for each test run was to start with one client and increase the number of clients over time. As noted above, this was done every 3 minutes, resulting in 180 samples per client count. The unicast defer update time in LibVNCServer was set to the same value as the multicast update interval, which was set to 10 milliseconds.

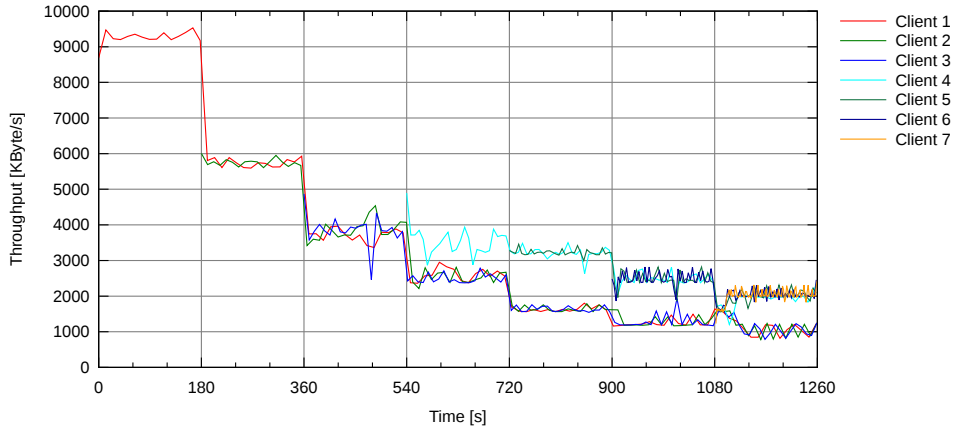


Figure 25: Per-client throughput of 1 to 7 clients achieved in a Fast Ethernet LAN with unicast VNC using Raw encoding. The plotted data is approximated using the Gnuplot 'acsplines' option with a weight of 1 for better readability.

The first test series measured achieved throughput in a LAN, with all machines being connected through a Gigabit Ethernet switch. The time-throughput diagrams of the unicast VNC and MulticastVNC results can be seen in figure 25 and figure 26, respectively. The graphs show test runs each lasting 21 minutes where an additional client would connect 180 seconds after its predecessor, as indicated by the vertical grid lines.

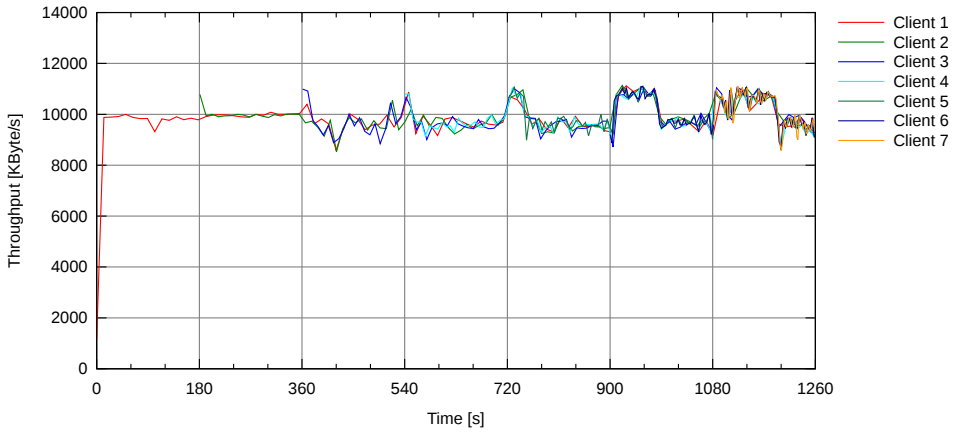


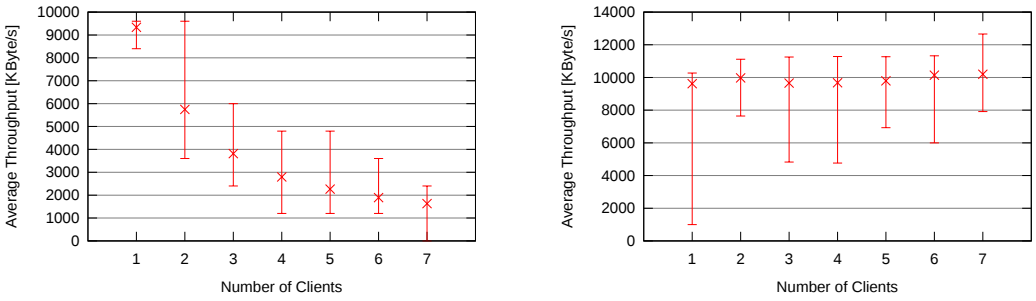
Figure 26: Per-client throughput of 1 to 7 clients achieved in a Fast Ethernet LAN with MulticastVNC using Raw encoding. The plotted data is approximated using the Gnuplot 'acsplines' option with a weight of 1 for better readability.

With traditional unicast data transmission (figure 25), per-client throughput decreases with each new client joining the session. Interestingly, clients 4 to 7 show a slightly different behaviour than clients 1 to 3, although the general trend is similar. Again, it is worth noting that the individual members of both groups show a tight correlation. The suspected reason for the differing characteristics is differences in the clients' TCP stacks: Clients 1 to 3 ran Linux whereas clients 4 to 7 had Windows XP installed.

In contrast to the unicast measurements, figure 26 shows that with MulticastVNC, per-client throughput is not as affected by the number of clients as it is when using traditional VNC. In fact, the graph shows that throughput seen by each client is around 10,000 KByte/s, independent of the number of clients in the session. What can be seen is that fluctuation seems to increase with more clients, this is possibly due to the MulticastVNC flow control. It is also worth noting that all clients expose similar behaviour in the multicast case, independent of the used operating system.

Figure 27 shows the above test series with averaged throughput, computed as the arithmetic mean of the samples taken by all active clients during the corresponding 180-second time span. The upper and lower ends of the error bars denote the biggest and smallest values sampled. In order to conserve space, all further experiments are discussed using graphs of this form only.

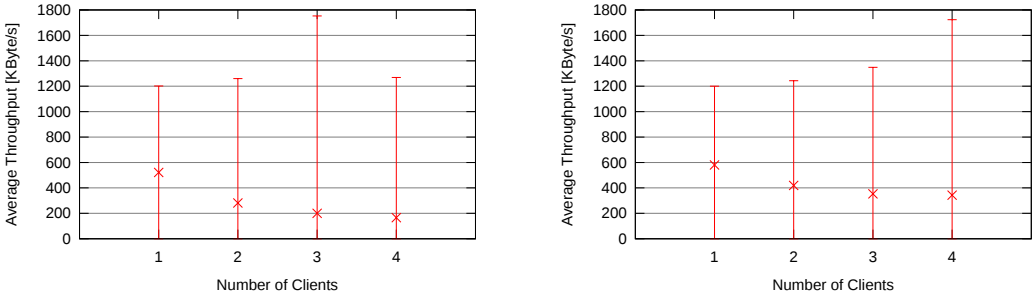
Another test series repeated the experiment in a WLAN. Due to lack of more WLAN adapters, these tests were conducted with 4 instead of 7 clients. These clients joined



(a) VNC. **(b)** MulticastVNC.

Figure 27: Average per-client throughput of 1 to 7 clients in a Fast Ethernet LAN using Raw encoding.

an ad-hoc wireless network created by the server machine. Since there were different generations of WLAN adapters in use on the client side, the server was configured to the 802.11b standard, the one used by the oldest client machine. This was done to ensure comparable measurements from all participating clients. Additionally, the server's wireless network adapter had to be set from the default automatic bit rate adaptation to a fixed rate in order to obtain proper multicast send rates[38]. The same issue was found with other WLAN adapters, it seems that many drivers use a default multicast send rate of 1 Mbit/s, regardless of the receiver's capabilities.



(a) VNC. **(b)** MulticastVNC.

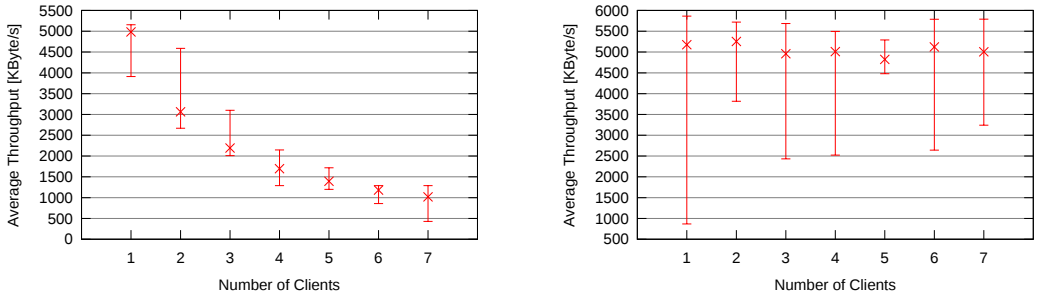
Figure 28: Average per-client throughput of 1 to 4 clients in a 802.11b WLAN using Raw encoding.

The corresponding unicast versus multicast graphs are shown in figure 28. It can be seen that in the unicast VNC case the achieved throughput decreases with an increasing number of clients, as in the LAN experiment. What is also notable is that the maximum observed value is higher than the maximum throughput actually achievable on this

[38] What is more, the procedures to change a WLAN adapter's multicast rate seem to differ wildly: Some can be set using the standard `iwconfig` to, some require setting internal variables with `iwpriv`, older ones come with their own tools.

link. The reason is to be sought in the way the MultiVNC client application samples throughput: the number of received bytes per second is calculated based on the number and dimensions of received RFB rectangles. These RFB rectangles can be dimensioned up to the size of the server's framebuffer. While this does not affect measurements taken in a high-throughput network, it has an effect on measurements in low-throughput networks: Here the transmission of a single big RFB rectangle can take several seconds, resulting in several value-0 samples and one relatively high one. While the effect averages out with an increasing number of samples, this can certainly be improved in the future.

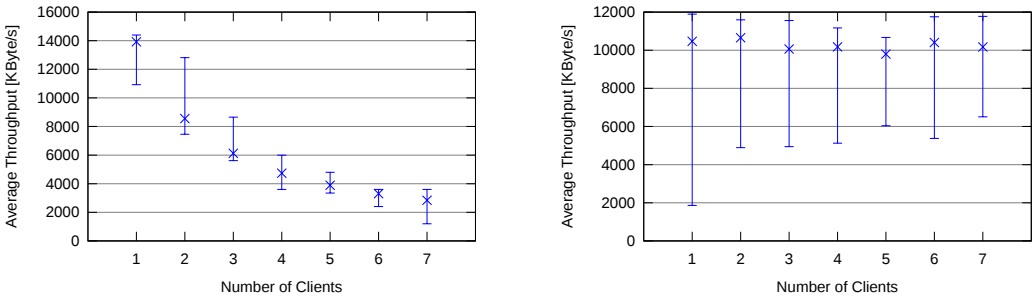
Although in the MulticastVNC case big RFB rectangles always are split up into smaller ones that fit into UDP datagrams, the effect is observable as well. The reason is that in the current implementation, a client's framebuffer is always initialized using traditional unicast VNC, even if all subsequent data transmission is multicast. Nonetheless, it can be seen in figure 28b that although the per-client throughput decreases with more clients as well, it does not do so as much as when using unicast VNC, in fact with 4 clients it is about twice as high. A possible cause for the observed decrease could be that with more clients, more *MulticastFramebufferUpdateRequest* messages are sent that interfere with the multicast traffic and cause packet loss. An interesting future prospect would be to repeat the experiment with more client machines to see if throughput would decrease more or if it would stay at around 400 KByte/s.



(a) VNC. **(b)** MulticastVNC.

Figure 29: Average per-client throughput of 1 to 7 clients in a Fast Ethernet LAN using Ultra encoding.

After VNC and MulticastVNC throughput characteristics were compared in LAN and WLAN environments using Raw VNC encoding, some tests were also done to see how Ultra VNC encoding would perform. Figure 29 shows the test results in a LAN. It can be seen that the throughput characteristics when using Ultra encoding are similar to the case where Raw encoding was used. The major difference is that average throughput is altogether lower. This is because achievable throughput is CPU bound in this configuration, the server CPU usage stats in figure 31 confirm this. With the relatively weak server machine used in the experiments, Ultra encoding performs similar to Raw encoding in terms of *decoded* data per second, as the graphs in figure 30 show. These also show that Ultra encoding roughly halves the amount of bytes being sent.



(a) VNC. **(b)** MulticastVNC.

Figure 30: Average decoded per-client throughput of 1 to 7 clients in a Fast Ethernet LAN using Ultra encoding.

Unfortunately, the Ultra VNC encoding tests could not be repeated in a WLAN because of a bug in the server's WLAN adapter driver. It seems the ath5k driver in use is not able to multicast small data packets (of around 600 Byte) for longer than roughly one minute. Thereafter, it complains about internal transmit queue overruns and completely stops sending. The same behaviour occurred when sending multicast traffic with the `iperf` tool. As a workaround, other server machines with different WLAN adapters (Intel PRO/Wireless 2100, Intel PRO/Wireless 3945ABG and Cisco WUSB54G) were tried out: The drivers for the Intel network adapters did not allow changing the multicast send rate, thus multicasting at only 1 Mbit/s. The Cisco adapter with the rt2500usb driver was plagued by the same symptoms as the ath5k driver. Under this circumstances, no measurements regarding Ultra encoded MulticastVNC traffic could be made. Although the issue could be worked around by only sending packets of a size the network adapter driver can handle, this does not tackle the root of the problem. It seems that multicasting in WLANs in general needs deeper investigation.

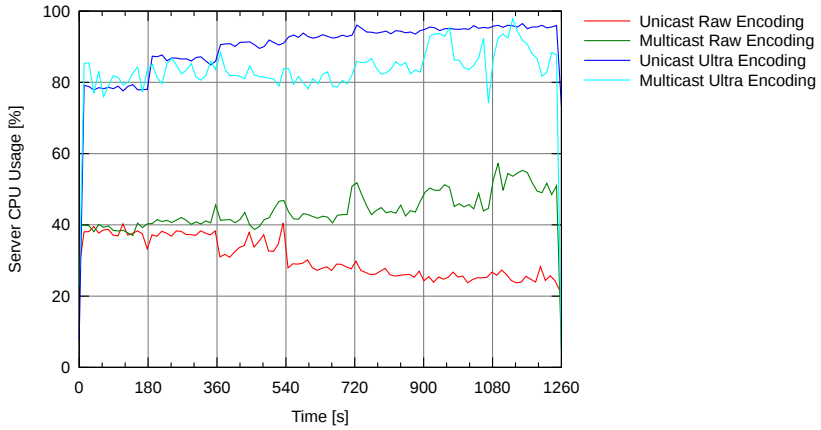


Figure 31: Server CPU usage for all Fast Ethernet LAN test runs.

Influence of the FastRequest Feature on Throughput

Another short test series sought to quantify the influence of the FastRequest feature built into the MultiVNC client application. This mode of operation was implemented to overcome performance issues of the RFB protocol on high-latency links and is explained in detail in section 4.2.3 on page 46.

Those experiments were composed of several test runs with either 1 or 4 clients that ran unicast VNC or MulticastVNC with FastRequest enabled or disabled. The eight tests resulting from the possible combinations were run in the Fast Ethernet LAN and 802.11b WLAN environments of the aforementioned tests and additionally in a Fast Ethernet LAN where a 150 ms delay was put onto the server's network interface using the netem emulation layer [32] to simulate a WAN. When FastRequest was enabled, it was set to an interval of 30 ms.

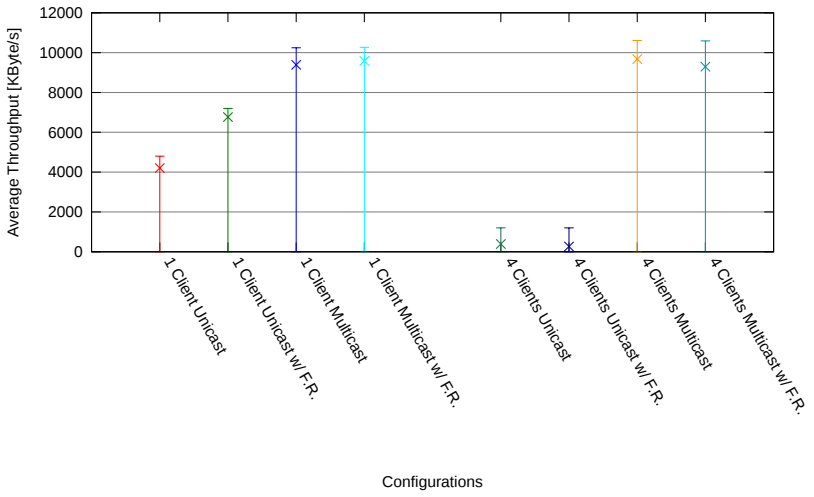


Figure 32: Per-client throughput of 1 and 4 clients, unicast and multicast, in a Fast Ethernet LAN with an emulated server-side delay of 150ms. Raw encoding was used.

The tests run in a LAN environment revealed a slight throughput increase for unicast VNC and virtually no change for MulticastVNC when FastRequest was enabled. In a WLAN, the feature had a slightly negative effect on throughput.

Only for the simulated WAN case significant effects could be observed, the results are shown in figure 32: When using traditional unicast VNC with one client, FastRequest was able to increase throughput from around 4000 KByte/s to circa 6800 KByte/s. On the other hand, multicast throughput of a single client was virtually unaffected and in fact higher than unicast throughput in both cases. This result was expected, as the MulticastVNC implementation used in the MultiVNC client constantly requests multicast framebuffer updates at the server's multicast update interval in order to drive the update process. As mentioned above, this update interval was set to 10 ms in all

test runs. Thus, more update requests were sent in the multicast case, resulting in more frequent updates from the server. When testing with 4 clients, the FastRequest feature had a slightly negative impact on throughput, probably because of the additional overhead involved.

Generally, the implementation of the FastRequest feature in MultiVNC still is improvable: Because sending and receiving of data are handled in the same thread, framebuffer update requests are not guaranteed to be sent at a constant rate. For instance while receiving a large block of data, nothing can be sent. An improved implementation as a future prospect would require fundamental changes concerning thread safety in the underlying LibVNCClient library.

Influence of Datagram Size on Multicast Throughput

Finally, some tests were conducted that investigated the effect of different multicast datagram sizes on throughput and packet loss. This was done with 1 and 4 clients on a Fast Ethernet LAN and a 802.11b WLAN with the maximum multicast datagram payload left at the default 1452 bytes or set to 2904, 5808, 11616, 23232, 46464 or 65487 bytes.

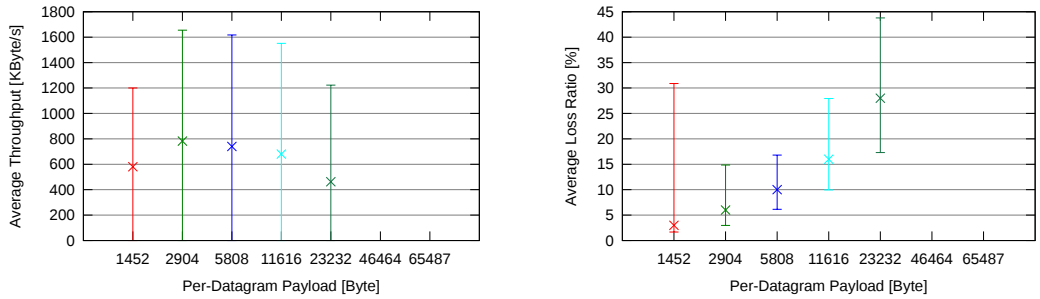


Figure 33: Average throughput and loss ratio of 1 client per different packet payloads in a 802.11b WLAN.

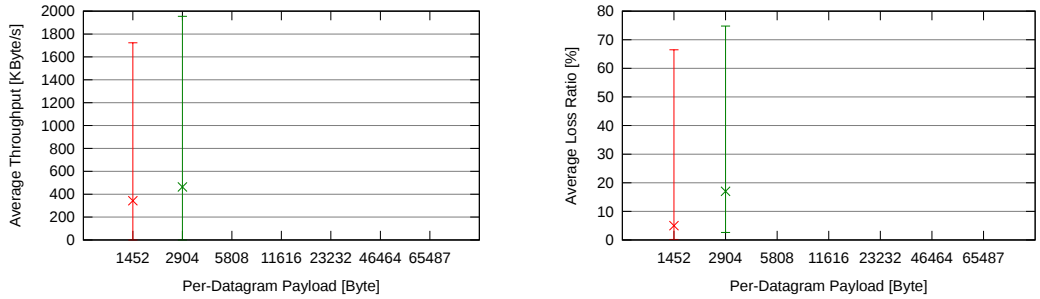


Figure 34: Average throughput and loss ratio of 4 clients per different packet payloads in a 802.11b WLAN.

The tests conducted in a LAN revealed that changing the per-datagram payload did not have any significant effect at all. Observed per-client throughput remained the same throughout all test runs, multicast loss ratio was virtually zero.

For MulticastVNC in a WLAN, results were more interesting: They showed that throughput increased when the default per-datagram payload of 1452 was doubled to 2904 bytes, as can be seen in figure 33. However, this also led to significantly increased average packet loss[39]. With higher datagram payload sizes, loss rates increased up to the point where no further measurements were feasible. Accordingly, achieved per-client throughput dropped as well. In sessions with 4 clients connected, the effect was even stronger: At a payload size of 5808 bytes loss rates were over 50% so that the experiment was cancelled (figure 34). It can thus be concluded that out of the tested maximum multicast datagram payload sizes 1452 byte is an optimal value.

Security Considerations

It must be mentioned that at the time of writing, all framebuffer updates sent to multicast clients are sent unencrypted. This certainly needs to be improved at a later time.

Comparison of Theoretical Predictions and Experimental Results

The expected throughput properties of unicast and multicast data transmission were formalized in section 2.2.2 on page 11 where the following metrics were presented:

For unicast data transmission, the maximum per-client throughput was expected to be

$$T_{cl} = \min\left(T_p, \frac{T_{sp}}{N_{sp}}\right) \tag{12}$$

where T_p would express the maximum throughput achievable on the network path p from server to client and T_{sp} describe the achievable throughput on the path sp, the subset of p that cl shares with $N_{sp} - 1$ other clients.

For multicasting, the maximum per-client throughput was predicted to be

$$T_{cl} = T_p \tag{13}$$

maintaining that the maximum throughput observable by cl is independent of the number of other clients it shares the network path to the server with.

The test results presented above in fact show that for unicast data transmission average per-client throughput decreases with an increasing number of connected clients. However, the measured values shown in figure 27a do not correspond exactly to the predicted

[39] The very high maximum loss rates stem from the MulticastVNC initialization phase: When a client joins a MulticastVNC session, there is a short time span where its multicast socket is already created and receiving while the client is busy with other initialization work. Thus, no data is read from the socket during this period, resulting in packets being dropped from the socket's receive buffer.

outcome of equation 12: Apparently, with a single client connected, the server is not able to fully saturate the link in a Fast Ethernet LAN. On the other hand, the other measurements shown in figure 27a seem to fit into the model described by equation 12: Each per-client throughput average multiplied by its corresponding client count gives a similar result of around 12,000 KByte/s. The reason for the deviation in the 1-client case most probably is that the unicast defer update time of 10 ms inhibits a higher average throughput. This assumption is supported by other tests where a 30 ms defer update time resulted in just 8800 KByte/s average throughput for a single client. The results of the 802.11b WLAN tests displayed in figure 28 do not exhibit this symptom since serving clients in this network environment takes a comparatively long time and thus the defer update time of 10 ms does not hamper achievable throughput. The WLAN results however also do not fully match the predictions made by equation 13. It is suspected that on this relatively error-prone transmission medium packet loss is interpreted by TCP as congestion and that thus a single TCP stream is not able to fully utilize available network resources.

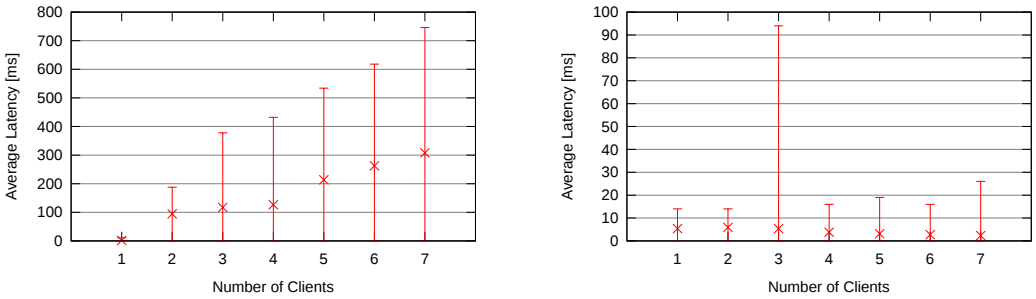
Regarding MulticastVNC, figure 27b shows that experimental results for a Fast Ethernet LAN match the theoretical predictions of the metric in equation 13: Average per-client throughput indeed seems to be independent of the number of connected clients. The 802.11b WLAN test results depicted in figure 28b however show a slight decrease in average throughput with an increasing number of connected clients. Achieved rates are still higher than in the unicast case for every number of clients. As mentioned above, a possible cause for this could be the circumstance that more framebuffer update requests are sent when more clients are connected, interfering with the multicast traffic and causing multicast packet loss. More in-depth examination of how MulticastVNC behaves in wireless networks seems a suitable topic for future work.

6.2.2 Latency Properties

The latency occurring for different configurations, i.e. a varying number of clients using unicast VNC or MulticastVNC in different network environments, was measured employing the same test methodology as was used in the throughput experiments: Each test run with a certain number of connected clients lasted 180 seconds, the unicast defer update time in LibVNCServer was set to the same value as the multicast update interval, 10 milliseconds.

Latency samples were obtained using the xvp-message-based mechanism built into the MultiVNC client application, which was introduced in section 5.1.3 on page 67 of this work. As explained there, the measured values reflect the data packet round trip time of the network in use *plus* the time the server takes to reply.

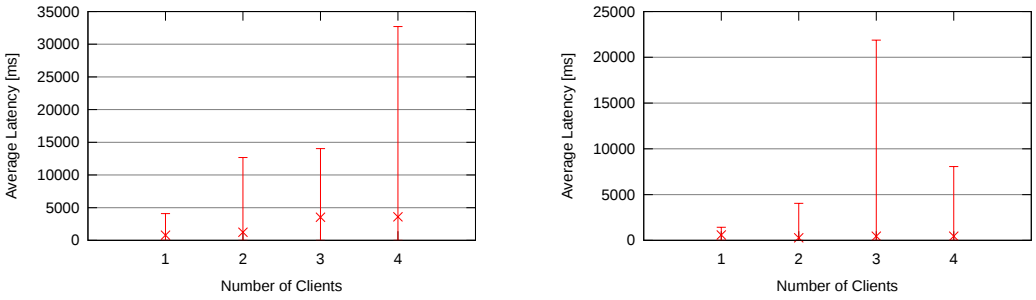
Figure 35 shows the results for 1 to 7 clients in a Fast Ethernet LAN using Raw VNC encoding. Again, the averages are computed as the arithmetic mean of valid samples taken by all active clients during the corresponding 180-second time span, the upper and lower ends of the error bars denote the biggest and smallest sampled values.



(a) VNC. **(b)** MulticastVNC.

Figure 35: Average per-client latency of 1 to 7 clients in a Fast Ethernet LAN using Raw encoding.

As can be seen in figure 35a, average per-client server answer time increases linearly with more clients to be served. The reason is that with more connected clients, an individual client faces a higher probability of having to wait for others to be served. With MulticastVNC, this problem does not exist because all clients are served at the same time, as shown by the graph in figure 35b: Here, average per-client server answer time stays at a constant low level, unaffected by the number of connected clients.

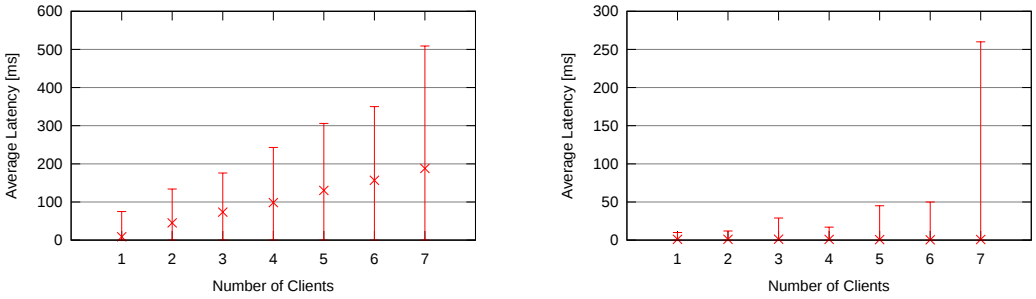


(a) VNC. **(b)** MulticastVNC.

Figure 36: Average per-client latency of 1 to 4 clients in a 802.11b WLAN using Raw encoding.

With the experiment repeated in an 802.11b WLAN with 1 to 4 clients (figure 36), the general pattern is similar: While for unicast VNC the average server answer time goes up with more connected clients, it remains at a constant level when MulticastVNC is used. Notable though are the very high maximum observed delays in the range of several seconds. These stem from the initialization phase when a client is about to join an existing session: Especially when there are other clients connected as well the first xvp reply from the server can take a very long time. The reason lies in the way the specific server implementation handles session initialization: In order not to disrupt the flow of data to other clients, the individual session initialization steps of a particular

client are interwoven with ongoing data transmissions of others. When there is lots of data to transmit over a low-throughput link such as WLAN, the whole initialisation phase can take very long to complete.



(a) VNC. **(b)** MulticastVNC.

Figure 37: Average per-client latency of 1 to 7 clients in a Fast Ethernet LAN using Ultra encoding.

Another test series investigated how latency would be affected by a change of the VNC encoding in use. Figure 37 shows the per-client server answer time of 1 to 7 clients using Ultra VNC encoding in a Fast Ethernet LAN: Again, per-client server answer time goes up when more clients are connected in the unicast case and stays at a constant level when MulticastVNC is used. The difference to the Raw VNC encoding results is that delays are altogether lower. This is a somewhat expected outcome since with Ultra encoding a framebuffer update amounts to less bytes being sent and thus taking a shorter time to complete.

Testing out MulticastVNC with Ultra encoding in a WLAN was hampered by the same network adapter driver issues mentioned in section 6.2.1. Therefore, no comparative tests with Ultra encoding could be carried out.

Comparison of Theoretical Predictions and Experimental Results

Regarding latency, the advantages of multicast data transmission were formalized in section 2.2.2 on page 11 by the following metrics: For unicast data transmission, latency observed by a client *cl* was predicted as

$$L_{cl} = L_p + t_{srv} * (N_{sp} - 1) \tag{14}$$

where L_p would describe the latency of the client's connection to the server, t_{srv} denote the time the server needs to serve a single client and N_{sp} be the number of clients (including *cl*) that share the same path to the server.

For multicast data transmission, per-client latency was expected to be

$$L_{cl} = L_p \tag{15}$$

because data is sent only once instead of N_{sp} times.

The test results indeed show that when unicasting, average per-client latency increases the more clients are connected. The linear increase shown in the diagrams above corresponds with equation 14. For multicast data transmission, test results showed a constant average per-client delay as predicted by equation 15. Insofar it can be maintained that the test results match the theoretical expectations and that specifically the MulticastVNC extension delivers the desired results.

6.2.3 Effectiveness of Multicast Flow Control

After the throughput and latency properties of the MulticastVNC extension were examined above, this section investigates how well the implemented multicast flow control scheme works. As mentioned in section 4.3.5, the MulticastVNC flow control builds upon related work done in [122] and [121], but makes two important modifications: First, instead of a single NACK it requires a burst of k NACK messages for a send rate decrease to occur. Second, it sets the send rate increment timer t to a fixed value instead of letting t depend on the current send rate. This section explains why these changes were necessary and shows that the resulting multicast flow control scheme is in fact working in both wired and wireless network environments.

Test Setup

The corresponding test runs all followed the same basic procedure: A client connected to the server and ran at full receive rate for 30 seconds. After that, it throttled its receive rate to circa 50% and ran in this configuration for another 30 seconds. On expiration of that time span, the client unthrottled again and ran like this for a final 30 second interval. During execution of these high-low-high profiles, the transmission rate of the server's network interface was observed to see how the server adapted its send rate to the respective new situation. Tests were carried out in both a Fast Ethernet LAN and a 802.11b WLAN.

The receive rate throttling on the client side was done using the netem emulation layer together with an Intermediate Functional Block pseudo-device as described in [32]: In such a setup, incoming traffic arriving at the client's network interface is redirected to an ifb pseudo-device, to which a rate-limiting token bucket output queueing discipline is attached. In the Fast Ethernet tests, the client's receive rate was throttled to 50 Mbit/s this way while in the 802.11b WLAN tests it was throttled to 4 Mbit/s.

Transmission rate logging on the server side was done using a simple script that calculated the number of sent bytes by examining `ifconfig` output every second.

Test Results

The first incarnation of the MulticastVNC flow control scheme was closely modeled after the original algorithm proposed in [122] and described in detail in section 4.3.5. The

parameter values of $m = 10$ and $n = 1.2$ used in [122] were adopted for the MulticastVNC flow control, a good choice for the c parameter though is left somewhat unclear in the paper. A value of $c = 100.000B$ was chosen after some testing in a LAN.

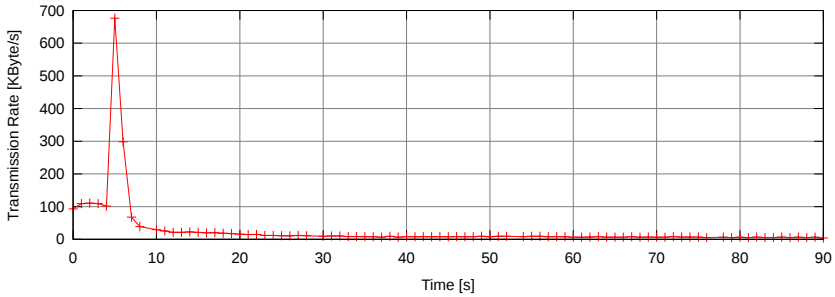


Figure 38: MulticastVNC server transmission rate in a 802.11b WLAN with $m = 10$, $n = 1.2$, $k = 1$, $T = \frac{100{,}000B}{R}$.

While this multicast flow control scheme worked reasonably well in a Fast Ethernet LAN, it failed completely when tested out in a WLAN, as shown by the diagram in figure 38: The rise in transmission rate at the beginning of the test stems from the client framebuffer initialization which is done via unicast, but then the server send rate is almost immediately throttled down to around zero. Since flow control worked well in a Fast Ethernet LAN before, it was suspected that NACKs generated from occasional packet loss in the WLAN were misinterpreted by the algorithm as a sign for receive buffer overflow at the client side, resulting in the server send rate to be lowered.

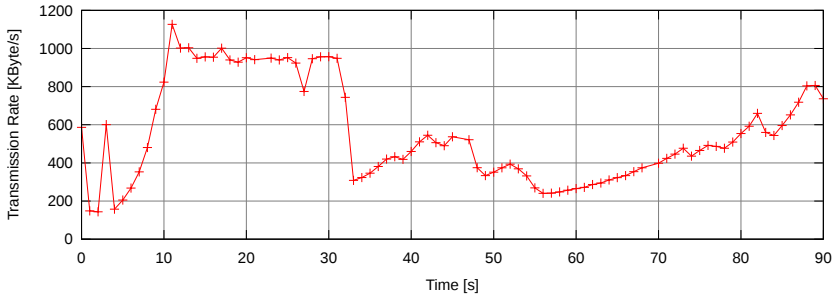


Figure 39: MulticastVNC server transmission rate in a 802.11b WLAN with $m = 10$, $n = 1.2$, $k = 3$, $T = \frac{100{,}000B}{R}$.

This misbehaviour could be fixed by changing the algorithm to only decrease the send rate upon receipt of a *burst* of NACKS. The reason this works better is that the patterns of NACKs generated by occasional packet loss and receive buffer overflow differ: In the former case, NACKs are mostly evenly distributed over time and tightly packed NACK bursts are relatively rare. However, on receive buffer overflow at the client side

a relatively large number of packets is dropped at once, resulting in a burst of NACK messages arriving at the server. A burst value of $k = 3$ was found to be adequate for both WLAN and LAN environments. Figure 39 shows that with the modification, the server send rate is not wrongly decreased in a WLAN anymore. However, it can be seen that the send rate increase after the client unthrottled at 60 seconds is way too slow.

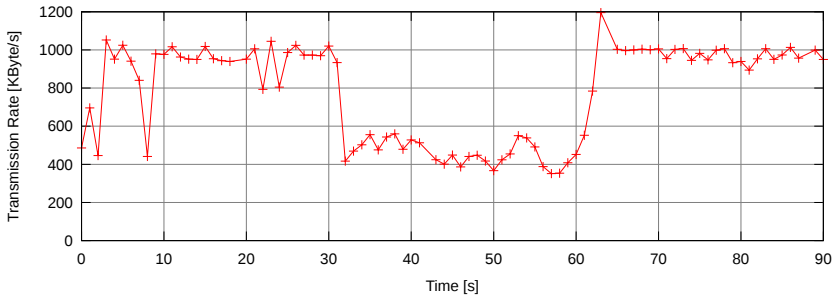


Figure 40: MulticastVNC server transmission rate in a 802.11b WLAN with $m = 10$, $n = 1.2$, $k = 3$, $T = \frac{25{,}000B}{R}$.

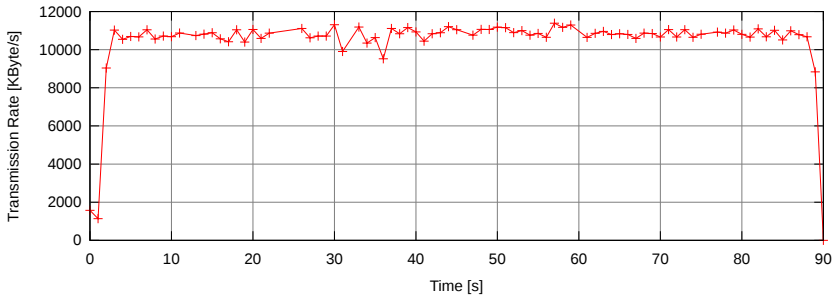


Figure 41: MulticastVNC server transmission rate in a Fast Ethernet LAN with $m = 10$, $n = 1.2$, $k = 3$, $T = \frac{25{,}000B}{R}$.

On this account, another pair of test runs was done with $c = 25.000B$, resulting in a lower send rate increment timer value. While this improved the server send rate adaptation response for the WLAN test (figure 40), this change made the flow control scheme fail in a LAN because the send rate was now increased too frequently (figure 41). In fact, no value for c could be found that would let the flow control work equally well in both Fast Ethernet LAN and 802.11b WLAN environments.

To solve this problem with either too frequent rate increases at high send rates or too slow send rate recovery at low send rates, a fixed send rate increment timer value t was introduced instead of the variable timer T that depended on the current send rate R. In fact, there actually is no reason for the send rate increment timer value to depend on the send rate itself. While it is true that more NACKs can be generated at a higher send rate

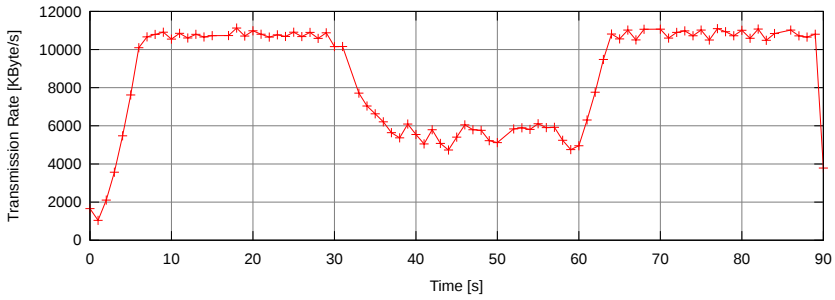


Figure 42: MulticastVNC server transmission rate in a Fast Ethernet LAN with $m = 10$, $n = 1.2$, $k = 3$, $t = 50ms$.

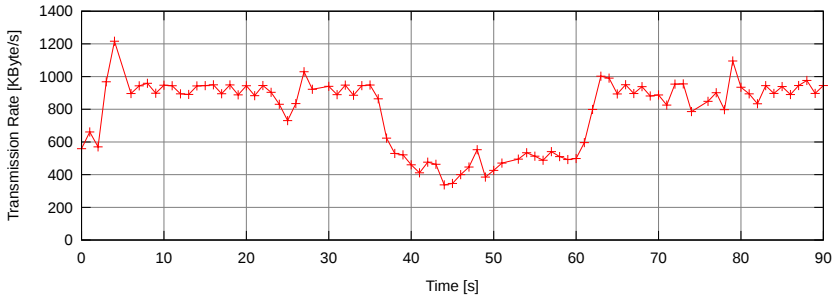


Figure 43: MulticastVNC server transmission rate in a 802.11b WLAN with $m = 10$, $n = 1.2$, $k = 3$, $t = 50ms$.

since more packets can be lost, this does not mean that the send rate has to be increased faster: The algorithm distinguishes between significant and meaningless NACKs and marks too high send rates as already decreased. This way, additional NACKs for a certain send rate have no effect.

Figures 42 and 43 show that the modified flow control scheme with NACK bursts and fixed send rate increment timer works well in both Fast Ethernet LAN and 802.11b WLAN, respectively. A parameter set of $m = 10$, $n = 1.2$, $k = 3$ and $t = 50ms$ was found to be adequate for both cases. These also are the values used in the throughput and latency experiments presented above. Further tuning of the flow control parameters is definitely possible but would need more in-depth testing whose documentation would go beyond the scope of this work. It would be promising topic for future work, though.

7 Summary and Future Prospects

As stated in section 1, this work had two main objectives: The first one was to create a computer-supported real-time collaboration system with support for fully concurrent multi-user operation. The second main objective was to design, implement and verify multicast data transmission support for this system called CollabKit.

The first objective could be achieved by connecting the Multi-Pointer-X extension MPX with the remote desktop technology VNC. This allows multiple users to interact on a shared remote desktop in a way which is similar to how they would operate a local desktop: Each participant is provided with an own mouse cursor and keyboard focus that are implemented at the window system level. Thus a standard desktop can be used concurrently by several remote users, operation is not confined to a limited set of applications that rely on a special groupware toolkit nor do users have to take turns.

Specific functionality required by the considered use cases was integrated as well: CollabKit allows per-user graphical annotations on the shared desktop, useful for instance in presentations or for collaboratively developing ideas. For electronic teaching and assistance as well as for professional remote collaboration, a window sharing feature was added that enables users to show local windows to others by exporting them to the remote desktop. The CollabKit client application can be used on Windows and Linux systems, a Mac OS X implementation exists, but needs more work.

The second focal point of this work was to fit the created system with support for multicast data transmission in order to make it perform well even with a high number of users in a low-throughput network. To accomplish this, an extension of the RFB protocol used by VNC was developed and implemented. This was done in a backwards compatible way by introducing new RFB encodings and corresponding new message types. Unlike other examined multicast-enabled remote desktop systems, CollabKit also implements multicast flow control and error handling using a NACK mechanism.

Experiments showed that compared to its unicast counterpart, the MulticastVNC extension performs significantly better when several client computers are connected to the system: While with unicast per-client throughput decreases with additional clients, the use of MulticastVNC makes average per-client throughput largely independent of the number of connected clients. Regarding latency, test results showed that in the unicast case average per-client delay goes up with an increasing number of connected clients while with MulticastVNC it stays at a constant low level. The integration of multicast data transmission into CollabKit thus had the desired effect of allowing a high number of users to still efficiently use the system even in a low-throughput network.

While CollabKit already is a usable shared view desktop conferencing system, there are further enhancements and feature additions conceivable: For instance, multi-user operation of the shared desktop currently lacks a fine-grained floor control scheme that goes beyond the simple full-control versus view-only access control that CollabKit uses. This could include a concept like window or application ownership where users can take exclusive control of a particular application, which can then be shared with others, passed

on or released. Productivity of collaboration could further be improved by integrating instant messaging and file transfer functionality into CollabKit.

Regarding the MulticastVNC extension of the RFB protocol, future work could focus on implementing other VNC encodings than Raw and Ultra or investigate possible encryption schemes for the pixel data sent via multicast. Another interesting topic could be to examine how multicast can be applied to the NACK mechanism so that lost datagrams are not necessarily retransmitted by the server but by other clients that have the requested data available. Also, future efforts can be directed at refining the implemented MulticastVNC flow control: A working set of parameters was found, but there is certainly room for improvement and motivation for further experiments here. In addition, how MulticastVNC behaves when used on the Internet remains to be tested out as well. Other more short term work includes improving the NACK mechanism on the client side, retransmission of lost multicast datagrams is only requested once currently. Also, the server implementation of how data packets are crafted is improvable: Sending compressed pixel data currently results in smaller datagrams, the configured maximum payload size is not fully utilised. Thus, there is unnecessary protocol overhead in this case. Generally, the experiences made with MulticastVNC in 802.11 wireless networks led to the realisation that multicasting at reasonable transmission rates is highly dependent on the network adapter driver in use: Most drivers seem to limit multicast traffic to a default rate of 1 Mbit/s and only some allow configuration of a higher rate. More in-depth investigation of this issue is certainly needed if the multicast data transmission feature of CollabKit is to be used on a wide range of WLAN hardware.

Summarizing the above statements, evaluation of the developed system showed that the two fundamental objectives set for this work could be achieved. Also, CollabKit was able to meet the particular requirements of the considered use cases.

It can be stated that by integrating existing technologies and extending them where needed, a computer-supported real-time collaboration system that supports concurrent multi-user interaction and multicast transmission of image data can indeed be created with reasonable effort.

List of Figures

List of Tables

References

[1] Adobe ConnectNow web site. `http://www.adobe.com/acom/connectnow/` – Retrieved: 2010-05-05

[2] Adobe ConnectPro web site. `http://www.adobe.com/de/products/acrobatconnectpro/` – Retrieved: 2010-05-05

[3] Apple Bonjour web site. `http://developer.apple.com/networking/bonjour/index.html` – Retrieved: 2010-06-26

[4] Apple Remote Desktop web site. `http://www.apple.com/remotedesktop/` – Retrieved: 2010-05-04

[5] Avahi project web site. `http://www.avahi.org` – Retrieved: 2010-06-23

[6] Cairo project web site. `http://www.cairographics.org/` – Retrieved: 2009-12-15

[7] Chromium project web site. `http://chromium.sourceforge.net/` – Retrieved: 2010-05-03

[8] Collaborative VNC web site. `http://benjie.org/software/linux/collaborative-vnc` – Retrieved: 2009-12-13

[9] Drawtop web site. `http://blogs.vislab.usyd.edu.au/moinwiki/DrawTop` – Retrieved: 2010-05-04

[10] EtherPad web site. `http://etherpad.org/` – Retrieved: 2010-07-30

[11] FreeNX project web site. `http://freenx.berlios.de/` – Retrieved: 2009-12-12

[12] Go-Global web site. `http://www.graphon.com/content/view/7/102/` – Retrieved: 2010-05-05

[13] Gobby web site. `http://gobby.0x539.de/` – Retrieved: 2010-07-30

[14] GoToMyPC web site. `https://www.gotomypc.com/` – Retrieved: 2010-05-04

[15] Gromit project web site. `http://www.home.unix-ag.org/simon/gromit` – Retrieved: 2009-12-12

[16] Gromit screenshot. `http://www.home.unix-ag.org/simon/gromit/screenshot2.jpg` – Retrieved: 2009-12-14

[17] GTK+ project web site. `http://www.gtk.org` – Retrieved: 2010-05-24

[18] GTK+ roadmap. `http://live.gnome.org/GTK%2B/Roadmap` – Retrieved: 2010-07-23

[19] iScribble web site. `http://www.iscribble.net` – Retrieved: 2010-07-30

[20] Libvncserver Git Repository. . – `http://libvncserver.git.sourceforge.net/` – Retrieved: 2010-02-05

[21] LibVNCServer project web site. `http://libvncserver.sourceforge.net/` – Retrieved: 2009-12-13

[22] libXcursor web site. http://cgit.freedesktop.org/xorg/lib/libXcursor/ – Retrieved: 2009-12-15

[23] libXi web site. http://cgit.freedesktop.org/xorg/lib/libXi/ – Retrieved: 2010-06-04

[24] libXtst web site. http://cgit.freedesktop.org/xorg/lib/libXtst/ – Retrieved: 2010-06-04

[25] LogMeIn web site. https://secure.logmein.com/DE/home.aspx – Retrieved: 2010-05-04

[26] Lotus Sametime Unyte Share web site. https://www.unyte.net/products/unyteshare.php – Retrieved: 2010-05-05

[27] MAST project web site. http://www.acet.reading.ac.uk/projects/mast/download.php – Retrieved: 2010-05-05

[28] MetaVNC web site. http://metavnc.sourceforge.net – Retrieved: 2009-12-13

[29] Microsoft SharedView web site. http://connect.microsoft.com/site94 – Retrieved: 2010-05-05

[30] Mikogo web site. http://www.mikogo.com/ – Retrieved: 2010-05-05

[31] MulticastVNC web site. http://www2.in.tum.de/~ziewer/multicastvnc/ – Retrieved: 2010-05-04

[32] Netem web site. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem – Retrieved: 2011-03-11

[33] NoMachine NX web site. http://nomachine.com/documentation/html/intr-technology.html – Retrieved: 2009-12-12

[34] Old MPX web site. http://wearables.unisa.edu.au/mpx – Retrieved: 2009-12-12

[35] PaintChat web site. http://www.paintchat.jp/ – Retrieved: 2010-07-30

[36] Rdesktop project web site. http://www.rdesktop.org/ – Retrieved: 2010-05-03

[37] RealVNC web site. http://www.realvnc.com/ – Retrieved: 2010-05-03

[38] RFB protocol, community version. http://tigervnc.org/cgi-bin/rfbproto – Retrieved: 2011-02-03

[39] Scriblink web site. http://www.scriblink.com/ – Retrieved: 2010-07-30

[40] SharedAppVNC web site. http://shared-app-vnc.sourceforge.net – Retrieved: 2009-12-13

[41] Sourceforge.net Software Search: "rdp client". http://sourceforge.net/search/?type_of_search=soft&words=%22rdp+client%22 – Retrieved: 2010-05-27

[42] Sourceforge.net Software Search: "vnc client". http://sourceforge.net/search/?words=%22vnc+viewer%22&type_of_search=soft&words=%22vnc+client%22&search=Search – Retrieved: 2010-05-04

[43] Spice project web site. http://www.spice-space.org/ – Retrieved: 2010-05-05

[44] SSVNC project web site. http://www.karlrunge.com/x11vnc/ssvnc.html – Retrieved: 2010-07-30

[45] SubEthaEdit web site. http://www.codingmonkeys.de/subethaedit/ – Retrieved: 2010-07-30

[46] Symantec PCAnywhere web site. http://www.symantec.com/norton/symantec-pcanywhere – Retrieved: 2010-05-05

[47] SynchroEdit web site. http://www.synchroedit.com/ – Retrieved: 2010-07-30

[48] Synergy web site. http://synergy2.sourceforge.net/ – Retrieved: 2010-05-04

[49] TeamViewer web site. http://www.teamviewer.com – Retrieved: 2010-05-05

[50] TeleTeachingTool (new implementation) web site. http://www2.in.tum.de/ttt – Retrieved: 2010-05-03

[51] TeleTeachingTool web site. http://ttt.uni-trier.de/ttt_download.html – Retrieved: 2010-05-04

[52] TightProjector web site. http://www.tightvnc.com/projector/ – Retrieved: 2010-05-02

[53] TightVNC project web site. http://www.tightvnc.com – Retrieved: 2009-12-13

[54] Timbuktu web site. http://www.netopia.com/software/products/tb2/ – Retrieved: 2010-05-04

[55] Twiddla web site. http://www.twiddla.com/ – Retrieved: 2010-07-30

[56] UltraVNC project web site. http://www.uvnc.com – Retrieved: 2009-12-13

[57] VeNCrypt project web site. http://sourceforge.net/projects/vencrypt – Retrieved: 2009-12-12

[58] Vino project web site. http://git.gnome.org/browse/vino/ – Retrieved: 2010-05-03

[59] Win2VNC project web site. http://win2vnc.sourceforge.net/ – Retrieved: 2009-12-12

[60] Windows Desktop Sharing web site. http://msdn.microsoft.com/en-us/library/aa373852.aspx – Retrieved: 2010-05-03

[61] Windows Meeting Space web site. http://www.microsoft.com/windows/windows-vista/features/meeting-space.aspx – Retrieved: 2010-05-03

[62] Windows Multipoint Mouse Mischief web site. http://www.microsoft.com/multipoint/mouse-mischief/default.aspx – Retrieved: 2010-05-15

[63] Windows Multipoint SDK web site. `http://www.microsoft.com/multipoint/mouse-sdk/` – Retrieved: 2010-05-15

[64] Windows Multipoint Server 2010 web site. `http://www.microsoft.com/windows/multipoint/default.aspx` – Retrieved: 2010-05-15

[65] Windows Server 2008 R2: Remote Desktop Services web site. `http://en.wikipedia.org/wiki/Remote_Desktop_Services#Windows_Desktop_Sharing` – Retrieved: 2010-05-03

[66] wxServDisc project web site. `http://sourceforge.net/projects/wxservdisc/` – Retrieved: 2010-06-04

[67] wxWidgets project web site. `http://www.wxwidgets.org/` – Retrieved: 2010-06-04

[68] x2x project web site. `http://x2x.dottedmag.net` – Retrieved: 2009-12-12

[69] xf4vnc project web site. `http://xf4vnc.sourceforge.net` – Retrieved: 2009-12-13

[70] XMX project web site. `http://www.cs.brown.edu/software/xmx` – Retrieved: 2009-12-12

[71] xrdp project web site. `http://xrdp.sourceforge.net/` – Retrieved: 2010-03-05

[72] xtv project web site. `http://www.ibiblio.org/pub/Linux/X11/xutils/xtv-1.1.tar.gz` – Retrieved: 2009-12-12

[73] Yuuguu web site. `http://www.yuuguu.com` – Retrieved: 2010-05-05

[74] Symantec PCAnywhere userguide. `ftp://ftp.symantec.com/public/english_us_canada/products/pcanywhere/12.0/manuals/pcauser.pdf` – Retrieved: 2010-05-15, 2006

[75] Adamson, B. ; Bormann, C. ; Handley, M. ; Macker, J.: RFC 3940 – Negative-acknowledgment (NACK)-Oriented Reliable Multicast. (2004)

[76] Adamson, B. ; Bormann, C. ; Handley, M. ; Macker, J.: RFC 5401 – Multicast Negative-Acknowledgement (NACK) Building Blocks. (2008)

[77] Albanna, Z. ; Almeroth, K. ; Meyer, D. ; Schipper, M.: RFC3171 – IANA guidelines for IPv4 multicast address assignments. (2001)

[78] Baecker, R. M.: *Readings in groupware and computer-supported cooperative work: Assisting human-human collaboration.* Morgan Kaufmann, 1993

[79] Baratto, R. A. ; Kim, L. N. ; Nieh, J.: Thinc: A virtual display architecture for thin-client computing. In: *ACM SIGOPS Operating Systems Review* 39 (2005), No. 5, p. 290

[80] Boyaci, O.: BASS Application Sharing System web site. `http://www.cs.columbia.edu/~boyaci/bass` – Retrieved: 2010-05-02

[81] Boyaci, O. ; Schulzrinne, H.: BASS Application Sharing System. In: *Tenth IEEE International Symposium on Multimedia, 2008. ISM 2008*, 2008, p. 432–439

[82] Cheshire, S.: Zero configuration networking (Zeroconf). `http://www.zeroconf.org` – Retrieved: 2010-05-03

[83] Coulthart, D. ; Das, S. ; Kim, L.: THINCing Together: Extending THINC for Multi-User Collaborative Support. (2009)

[84] Dix, A. ; Finlay, J. ; Abowd, G. D.: *Human-computer interaction*. Prentice Hall, 2004

[85] Drake, K. ; Ltd, U. S.: XTEST Extension Protocol. In: *X Consortium Standard* (1994)

[86] Ellis, C. A. ; Gibbs, S. J. ; Rein, G.: Groupware: some issues and experiences. In: *Communications of the ACM* 34 (1991), No. 1, p. 39–58

[87] Gemmell, J. ; Montgomery, T. ; Speakman, T. ; Crowcroft, J.: The PGM reliable multicast protocol. In: *Network, IEEE* 17 (2003), No. 1, p. 16–22

[88] Gokhale, A. A.: Collaborative learning enhances critical thinking. In: *Journal of Technology Education* 7 (1995), p. 22–30

[89] Gross, T. ; Koch, M.: *Computer-supported cooperative work*. Oldenbourg Wissenschaftsverlag, 2007

[90] Gutwin, C.: The effects of network delays on group work in real-time groupware. In: *ECSCW 2001*. Springer, 2001 (ECSCW), p. 299–318

[91] Hasan, S. M. ; Lewis, G. J. ; Alexandrov, V. N. ; Dove, M. T. ; Tucker, M. G.: Multicast Application Sharing Tool for the Access Grid Toolkit. In: *UK e-Science All Hands Meeting, Nottingham, UK*, 2005

[92] Hayden, M. ; Xiao, Z.: *A Multicast Flow Control Protocol*. Cornell University, 1999

[93] Hinden, R. ; Deering, S. et al.: RFC 2373 – IP version 6 addressing architecture. (1998)

[94] Hubinette, F.: x2vnc project web site. `http://fredrik.hubbe.net/x2vnc.html` – Retrieved: 2009-12-12

[95] Hutterer, P.: MPWM project web site. `http://cgit.freedesktop.org/~whot/mpwm/` – Retrieved: 2009-12-12

[96] Hutterer, P. ; Thomas, B. H.: Groupware support in the windowing system. In: *Proceedings of the eight Australasian conference on User interface-Volume 64*, 2007, p. 46

[97] ITU-T: ITU-T T.128 - Multipoint application sharing. (2008)

[98] Johansen, R.: Teams for tomorrow. In: *Proc. 24th IEEE Hawaii Intl Conf. on System Sciencs*, p. 520–534

[99] Johnson-Lenz, P. ; Johnson-Lenz, T.: Groupware: The process and impacts of design choices. In: *Computer-Mediated Communication Systems*. Academic Press, 1982

[100] Kaisler, Lora K.: Presentation clipart taken from http://21cif.com with kind permission of the author. `http://21cif.com/tutorials/micro/mm/activelinks/images/presentation.gif` – Retrieved: 2011-03-28

[101] Koifman, A. ; Zabele, S.: RAMP: A reliable adaptive multicast protocol. In: *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE* Vol. 3 IEEE (Conf.), 1996, p. 1442–1451

[102] Kotonya, G. ; Sommerville, I.: *Requirements engineering*. Wiley Chichester, 1998

[103] Lin, J. C. ; Paul, S.: RMTP: A reliable multicast transport protocol. In: *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE* Vol. 3 IEEE (Conf.), 1996, p. 1414–1424

[104] Marjanovic, O.: Learning and teaching in a synchronous collaborative environment. In: *Journal of Computer Assisted Learning* 15 (1999), p. 129–138

[105] Mark, G. ; Grudin, J. ; Poltrock, S. E.: Meeting at the desktop: An empirical study of virtually collocated teams. In: *ECSCW'99*. Springer, 1999, p. 159–178

[106] Meyer, D.: RFC 2365 – Administratively scoped IP multicast. (1998)

[107] Microsoft Corporation: Remote Desktop Protocol: Basic Connectivity and Graphics Remoting Specification. (2007). – `http://download.microsoft.com/download/9/5/e/95ef66af-9026-4bb0-a41d-a4f81802d92c/%5BMS-RDPBCGR%5D.pdf` – Retrieved: 2011-01-25

[108] Netopia Inc.: Timbuktu Datasheet. (2003). – `http://www.netopia.com/software/products/tb2/timbuktu_DS.pdf` – Retrieved: 2010-05-04

[109] Ng, Choon J. ; Takatsuka, Masahiro ; Smith, Steve ; Lowe, Nick: VNCast web site. `http://wiki.vislab.usyd.edu.au/moinwiki/VNCast` – Retrieved: 2010-05-06

[110] Packard, K.: XDMCP Specification. . – `http://www.x.org/docs/XDMCP/xdmcp.pdf` – Retrieved: 2009-12-14

[111] Packard, K. ; Gettys, J.: X Window System network performance. In: *FREENIX Track, 2003 Usenix Annual Technical Conference*, 2003

[112] Pingali, S. ; Towsley, D. ; Kurose, J. F.: A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In: *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems* ACM (Conf.), 1994, p. 221–230

[113] Prante, T. ; Magerkurth, C. ; Streitz, N.: Developing CSCW tools for idea finding-: empirical results and implications for design. In: *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, 2002, p. 106–115

[114] Rhee, I. ; Balaguru, N. ; Rouskas, G. N.: MTCP: Scalable TCP-like congestion control for reliable multicast. In: *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* Vol. 3 IEEE (Conf.), 1999, p. 1265–1273

[115] Richardson, T.: *The RFB Protocol V3. 8*. RealVNC Ltd., 2009

[116] Runge, K.: x11vnc project web site, `http://www.karlrunge.com/x11vnc` – Retrieved: 2009-12-13

[117] Schulzrinne, H. ; Casner, S. ; Frederick, R. ; Jacobson, V.: RFC 3550 – RTP: A transport protocol for real-time applications. (2003)

[118] Stahl, G. ; Koschmann, T. ; Suthers, D.: Computer-supported collaborative learning: An historical perspective. In: *Cambridge handbook of the learning sciences* (2006)

[119] T., Masahiro ; Lowe, N.: *DrawTop*. ViSLAB School of Information Technologies, University of Sydney, 2006

[120] Teufel, S. ; Sauter, C. ; Mühlherr, T. ; Bauknecht, K.: *Computerunterstützung für die Gruppenarbeit*. Addison-Wesley, 1995

[121] Yamamoto, K. ; Sawa, Y. ; Yamamoto, M. ; Ikeda, H.: Performance evaluation of ACK-based and NAK-based flow control schemes for reliable multicast. In: *TENCON 2000. Proceedings*, 2000, p. 341–345

[122] Yamamoto, M. ; Sawa, Y. ; Shinji, F. ; Ikeda, H.: NAK-based flow control scheme for reliable multicast communications. In: *Global Telecommunications Conference, 1998. GLOBECOM 98. The Bridge to Global Integration. IEEE*, 1998, p. 2611–2616

[123] Yang, S. J. ; Nieh, J. ; Selsky, M. ; Tiwari, N.: The performance of remote display mechanisms for thin-client computing. In: *Proceedings of the 2002 USENIX Annual Technical Conference* Vol. 6, 2002

[124] Ziewer, P. ; Seidl, H.: Transparent teleteaching. In: *Proceedings of ASCILITE*. 2002, p. 749–758

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ort/Datum Unterschrift

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Ort/Datum Unterschrift